simms

How to fast track the development of vision-based AI at the edge

Whitepaper

Abstract

As demand rises for real-time, intelligent vision applications from smart factories to autonomous drones the need to deploy AI (and its subset ML) at the edge has never been greater.

However, many software development teams face a steep learning curve when translating powerful AI models into responsive, power-efficient edge solutions. The challenges lie not only in model optimisation but also in navigating fragmented hardware ecosystems, achieving tight performance goals and ensuring scalability from prototype to production.

This white paper addresses the core pain points software engineers encounter when bringing vision-based AI to edge systems. Many of these pain points are as a result of the selected underlying hardware, and we explore in detail the strengths and weaknesses of the software stacks depending on which hardware is selected.

We also explore how adopting vision and AI-ready pre-validated platforms can dramatically accelerate time-to-market, simplify development and de-risk deployment, citing Innodisk's APEX-X100 platform by way of example.

Index

Introduction

The shift from cloud-centric artificial intelligence (AI) to the edge in vision-based systems has numerous benefits, including fast (up to real-time) decision making and enhanced privacy.

As for applications, there are many. For example, AI-enabled vision-based systems are appearing in industrial automation and manufacturing, automotive and transportation, retail and smart stores, agriculture, security and surveillance, aerospace and drones, and smart cities and homes.

For software engineers developing AI-enabled functionality the move to the edge introduces a number of significant technical challenges. For instance, unlike cloud-based systems with virtually unlimited resources, edge environments are heterogeneous, resource-constrained and often difficult to scale. Also, there is the familiar backdrop of commercial pressure to be early (and ideally first) to market with a high performance, reliable solution that serves a market need. Regulatory compliance is almost always required too.

Most of the technical challenges software engineers face have their roots in hardware: and we cannot stress this enough. For instance, depending on key performance requirements, vision-based AI at the edge apps can run on a variety of hardware device types, each with their own strength and weakness.

Devices with limited processing power and memory makes running deep learning models (such as convolutional neural networks, CNNs) difficult without model optimisation, a task that falls squarely with the project's software engineers.

Many edge environments vary significantly, particularly those outdoors (e.g. variable / poor lighting), there may also be motion-blur and, if the camera is on a moving platform, such as an autonomous guided robot (AGR), angles will be changing all the time. The upshot: models trained in the lab may underperform in the real world and, again, it is for the software engineer to find optimal solutions.

Another challenge is often data fusion, as some vision-based AI applications must work with multiple vision/sensing techniques (see figure 1). GPS may also be required, all of which means keeping a very tight control over time synchronisation.

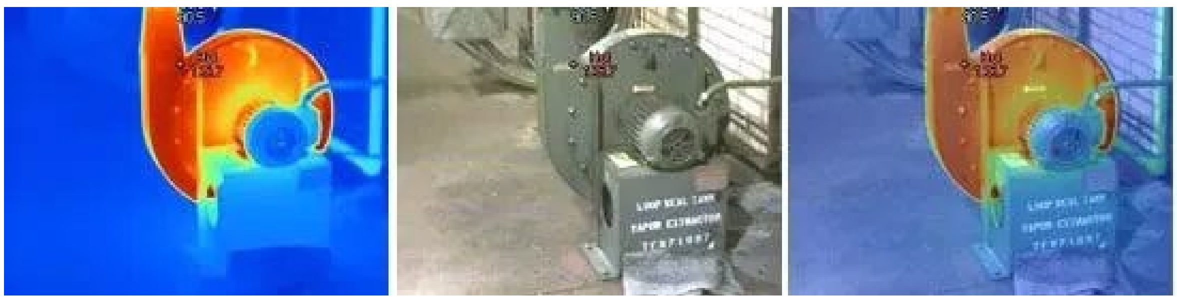

Figure 1. Image fusion (such as layering the images produced by visible light and thermal cameras) has been popular in industry for several years. AI/ML's role is to make sense of the data for automated predictive maintenance purposes, for example, and to perform object recognition (including the movement of personnel). Source: https://www.fluke.com/en-us/learn/blog/thermal-imaging/how-patent-pending-technology-blends-thermal-and-visible-light

Also, while the underlying hardware may provide some security features, the edge device might operate in a location where it is susceptible to tampering, data breaches and model theft.

Lastly, as the raison d’etre of edge processing is to have little if any reliance on the cloud, how easy is it to test, debug and upgrade (or if necessary, rollback) products in the field? Unless post-deployment considerations are factored in at the design stage, the edge device might have a very short life in the field.

Return to Index ^^

The Flow

As readers will be aware, there is a logical flow (see Figure 2) taken by software engineers once the hardware has been selected: and we must stress that it is important for the software team to be involved in that selection.

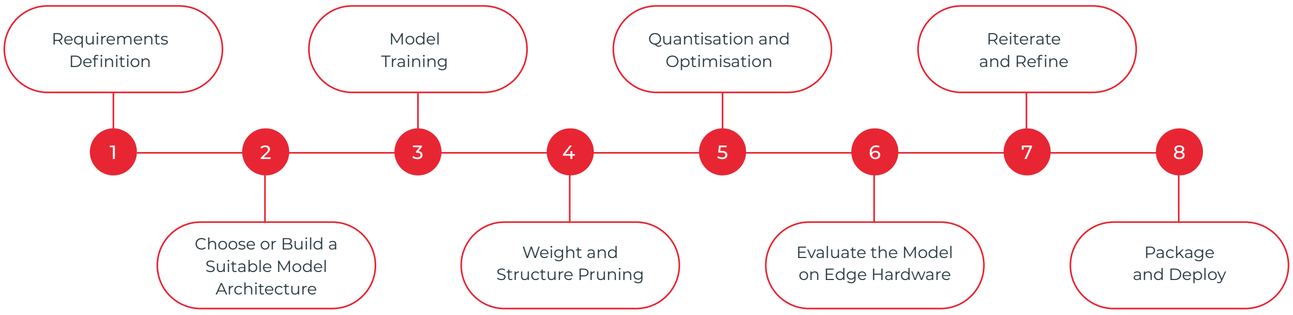
Requirements Definition
Choose or Build a Suitable Model Architecture
Model Training
Weight and Structure Pruning
Quantisation and Optimisation
Evaluate the Model on Edge Hardware
Reiterate and Refine
Package and Deploy
1
2
3
4
5
6
7
8

Figure 2: Developing a vision-based AI at the edge product requires following a flow, noting that parts are iterative and steps back may need to be taken.

1. Requirements Definition

Identify the tasks (e.g., object detection and classification), specify constraints (e.g., latency, power, memory and accuracy) and understand the strengths and weakness of the hardware. It is also essential that post-deployment considerations be included at this stage.

2. Choose or Build a Suitable Model Architecture

If selecting, there are several open-source ones to choose from including YOLOv8-Nano (part of the YOLOv8 model family developed by Ultralytics, and with code and models publicly available on GitHub) for detection. And for classification, an example model architecture is MobileNetV2, developed by Google as part of its TensorFlow ecosystem.

Tip
Model selection should involve a review of licensing terms. For example, while a General Public License (GPL) is common for most models, sometimes a GNU Affero GPL (AGPL) might be required. It is an extension to a standard GPL and ensures that the source code of any modified software to be used over a network can be made available to users interacting with it remotely. This means that if you modify your AGPL-licensed code (model) and run it as a service, users of your service are entitled to receive the source code.

3. Model Training

High quality datasets should be used that represent the deployment environment and the data should be augmented to train for conditions such as low-light and motion blur. Tip: to fast track the development of your application use transfer learning, if possible - i.e., take a model trained on one task and modify it to perform a different but related task.

4. Weight and Structure Pruning

Set low-importance weights to zero and remove any unnecessary filters and neurons. These pruning exercises will reduce model size/complexity and simplify computation tasks. However, be mindful that pruning comes at the cost of system accuracy.

5. Quantisation and Optimisation

Having trained and pruned the model it is now time to start preparing for edge hardware. Quantisation converts the model developed on a desktop machine at floating-point 32-bit resolution (FP32) to FP16, 8-bit integer (INT8) or another low-precision format. Quantisation-aware training (QAT) helps preserve accuracy. Also, now is the time to match the quantisation format with hardware support. For instance, Coral Edge TPU requires 8-bit quantised input tensors. The model is converted to the supported format of the target platform: e.g. OpenVINO IR (for Intel-based edge devices), TFLite (Android, Edge TPU), TensorRT (NVIDIA Jetson) or ONNX (cross-platform).

6. Evaluate the Model on Edge Hardware

Test for inference latency, memory usage, power consumption and accuracy versus the original model. Note: many vendor SDKs (e.g., NVIDIA Nsight, Intel VTune, Android Profiler) include useful profiling tools.

7. Reiterate and Refine

If too much accuracy is lost or latency is too high, steps 3 to 6 should be repeated. Accuracy improvement measures include adjusting pruning/quantisation parameters or using QAT (if not used in the first pass). Or even start with a better base model. Latency can be shortened through further pruning and quantising. Again, it's a balancing act between accuracy and speed.

8. Package and Deploy

As a minimum this involves bundling together the optimised model with the edge software stack. Also, if the ability to update in the field is a requirement (which probably will), over the air (OTA) updates must be enabled.

As mentioned, post-deployment considerations must be included as a part of the requirements definition. These considerations include:

- Edge systems must maintain reliable performance under varying environmental and network conditions. In addition, the pipeline (from image capture through to inference) may need to accommodate varying image quality (resolution, frame rates and encoding, etc).
- Real-time monitoring, remote fault recovery and OTA updates are as critical as model optimisation.
- For safety-critical applications, such as industrial monitoring or human-machine interaction, model integrity, data security, and update traceability must be built into the system from the start, as the information needs to be readily throughout the system's life in the field.

Return to Index ^^

Hardware Considerations

When developing software for a vision-based AI at the edge system in addition to understanding the functional requirements (objectives) it is important to appreciate the underlying hardware as it will impose restrictions on the software stack.

Let's start by considering the top-level implications of using the different kinds of hardware. We will look at each in detail shortly, but here is a brief comparison showing perhaps their main strengths and weaknesses where vison-based AI at the edge is concerned.

Device	Description	Main Strength	Main Weakness
Graphics Processing Unit (GPU)	A highly parallel processor designed for rapid image rendering and data-intensive tasks like deep learning.	Excellent parallel processing power for large-scale AI inference.	High power consumption and thermal output, making it less ideal for low-power edge environments.
Neural Processing Unit (NPU)	A dedicated AI accelerator optimised for executing neural network operations efficiently.	Great performance and energy efficiency for deep learning inference tasks.	Limited flexibility. NPUs primarily support specific AI workloads and lack general-purpose capabilities.
Central Processing Unit (CPU)	A general-purpose processor capable of handling diverse computing tasks including control logic and OS management.	Versatile and essential for managing system-level operations and pre/post-processing in AI pipelines.	Poor parallelism and slower performance on deep learning workloads compared to dedicated accelerators.
Digital Signal Processor (DSP)	Optimised for real-time signal processing tasks such as filtering, FFTs, and low-level vision algorithms.	Efficient for low-latency, real-time signal processing with low power usage.	Limited performance on modern deep learning models and complex vision tasks.
Field-Programmable Gate Array (FPGA)	A reconfigurable hardware device that can be programmed to implement custom logic for specific tasks.	High flexibility and parallelism tailored to application-specific acceleration.	Complex to program and optimise, with longer development cycles compared to fixed-function accelerators.
Microprocessor Unit (MPU)	A general-purpose processor used in embedded systems and capable of running operating systems and managing complex applications.	Supports full operating systems and multitasking in moderately resource-constrained environments.	Lacks native AI acceleration.

Device	Description	Main Strength	Main Weakness
Microcontroller Unit (MCU)	A compact processor with tightly integrated memory and peripherals, designed for simple control tasks in embedded systems.	Ultra-low power consumption and simplicity for lightweight edge devices.	Limited processing power and memory for running vision-based AI models.
Tensor Processing Unit (TPU)	A specialised AI accelerator developed by Google to perform tensor operations used in neural networks.	High throughput and energy efficiency for running large neural network inference workloads.	Limited flexibility, with support focused mainly on TensorFlow and specific model architectures.

Above, only one strength and one weakness was shown for each type of hardware. The following sections go into more detail, plus we discuss the software stacks, programming languages and frameworks. Alternatively, you can skip these sections and go straight to Vision-Ready SOMs.

Return to Index ^^

GPU

Vision-based AI at the edge relies heavily on deep neural networks, especially convolutional CNNs, which require large amounts of matrix and vector computations. GPUs lends themselves well to CNN tasks thanks to their massive parallel processing capabilities.

Specifically, they have thousands of cores that can perform operations in parallel, accelerating inference workloads significantly. This makes them ideal for handling the computational workload of deep learning tasks.

GPUs exhibit low latency, low enough to support real-time inference and for processing high-resolution video streams and are well supported by major AI frameworks (TensorFlow, PyTorch, ONNx etc.) and many edge AI SDKs have optimised software stacks for GPU acceleration.

GPU Software Stack Strengths

- Optimised AI Frameworks. For example, TensorRT, cuDNN and DeepStream SDKs enable efficient deployment of DNNs with quantisation, pruning and layer fusion.
- Containerised Environments. Tools like Docker and NVIDIA NGC make it easy to deploy consistent environments at the edge.
- Hardware-Software Integration. Tight integration between hardware (Jetson Xavier, Orin, etc.) and software (JetPack SDK) enables full utilisation of GPU capabilities.
- Model Optimisation. Support for FP16/INT8 quantisation and pruning to improve performance while reducing memory footprint.
- Ecosystem and Community. Strong developer support, documentation, and ecosystem around NVIDIA and CUDA-based stacks.

GPU Software Stack Weaknesses

- Portability Issues. Models optimised for GPUs (using TensorRT, CUDA) are often not portable across non-NVIDIA hardware.
- High Complexity. Full deployment pipelines (training, conversion, optimisation, deployment) are complex and require specialised knowledge.
- Limited Framework Support. Not all new model architectures are supported out of the box by TensorRT or other GPU inference tools.
- Integration Overhead. Integrating GPU inference into embedded systems may require custom drivers, tuning and/or middleware.

GPU Programming Languages

Language	Usage	Pros	Cons
CUDA (C/C++)	Core language for programming NVIDIA GPUs.	Full control over GPU memory and kernels. Maximum performance.	Steep learning curve. Vendor lock-in (NVIDIA only). Complex debugging and profiling.
Python (via PyTorch, TensorFlow)	High-level AI development. GPU usage abstracted via backends (CUDA/cuDNN).	Easy and fast prototyping. Large community and support.	Slower than C++ in real-time inference. Less control over low-level optimisation.
OpenCL (C-based)	Portable parallel programming across vendors (NVIDIA, AMD, Intel).	Cross-vendor compatibility.	Less optimised than CUDA on NVIDIA GPUs.
C++	Often used for deploying optimised applications with TensorRT or OpenCV.	High performance and low-level control.	Verbose and requires more development time than Python.
Rust	Emerging as an AI language, and attractive for applications where there is a safety focus.	Comparable to C/C++ but considered safer. Great for low-latency, high-throughput workloads.	Steep learning curve. Small (but growing) ecosystem and community.

Example Frameworks for Vision-Based AI at the Edge on GPUs

Framework	Use Case	Pros	Cons
NVIDIA JetPack SDK	Full-stack edge AI on Jetson.	Optimised for Jetson. Integrated CUDA/cuDNN/TensorRT.	NVIDIA-only. Steep learning curve.
TensorRT	High-performance inference.	Fast inference. Supports quantisation (INT8/FP16). ONNX model import.	Complicated API. NVIDIA-only.
DeepStream SDK	Video analytics at the edge.	High throughput. Optimised pipelines. GStreamer integration.	Complex configuration. Limited flexibility.
PyTorch + TorchScript	Training and deployment.	Easy to use. Export to ONNX. GPU acceleration via CUDA.	Slower than TensorRT. More RAM needed.
TensorFlow Lite + GPU Delegate	Mobile and edge inference.	Small binary size. Cross platform.	GPU support weaker than NVIDIA stack.
ONNX Runtime (with TensorRT backend)	Inference across devices.	Converts PyTorch/TensorFlow models. Runs on GPU (via TensorRT).	Compatibility issues with custom operations.
OpenCV + CUDA Modules	Vision pre/post-processing.	Real-time image processing. Runs on GPU.	Not AI-specific. Manual optimisation needed.

To accelerate the development of AI-enabled applications some GPU vendors offer a great deal of support. Worthy of particular note is NVIDIA's CUDA-X AI (see Figure 3), a complete deep learning programming model and software stack for researchers and software developers to build high performance GPU-accelerated applications for, amongst other things, computer vision.

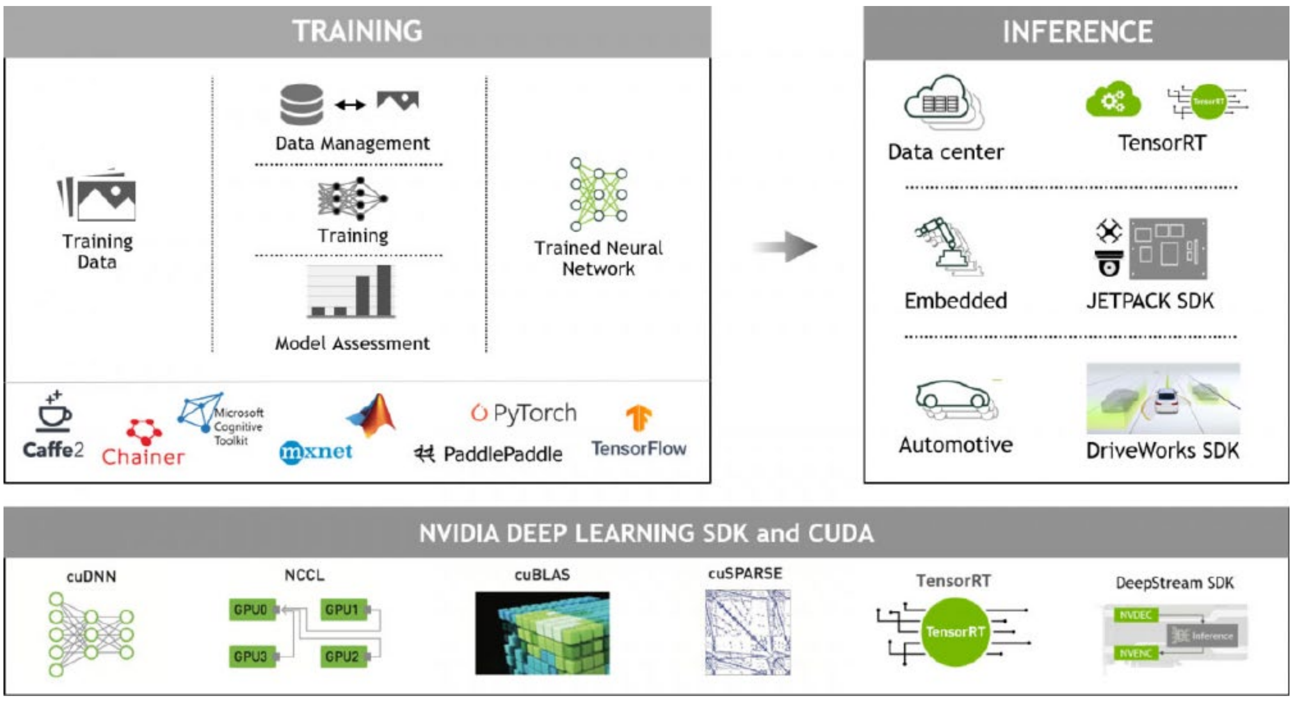

Figure 3. Built on CUDA-X, NVIDIA's unified programming model provides a way to develop deep learning applications on the desktop or data centre, and deploy them to resource-constrained IoT devices. Source https://developer.nvidia.com/deep-learning-software

Note:
Also worthy of note - but something the hardware team should already be aware of - is that not all GPUs include dedicated hardware coders/decoders (CODECs). Most NVIDIA GPUs do: their CODECs are called NVENC and NVDEC). If dedicated hardware CODECs are not present within the selected GPU this will impact the ability to record or stream video directly from the device – e.g., inference plus digital video record (DVR) functions. The knock-on effect of poor hardware choice can lead to increased latency, excessive CPU load and reduced power efficiency.

Return to Index ^^

NPU

NPUs are proving increasingly popular in vision-based AI at the edge applications such as drones, autonomous vehicles and smart sensors. They are optimised for matrix and tensor operations (core to neural networks) and boast high throughput and low power consumption.

They are good at parallelism and can process multiple computations concurrently, which is vital for CNNs. In addition, NPUs are low latency and can be used for real-time inference, essential for object detection, face recognition and autonomous navigation, for example. They are energy efficient, too, so are ideal if there is a tight power budget.

NPU Software Stack Strengths

These include the fact that many NPUs come with optimised AI libraries - from vendors like Qualcomm, Google, ARM and Intel, for instance – that provide pre-compiled operations, quantisation tools and compilers to boost performance. Most NPUs also support popular ML frameworks and software stacks often include tools for 8-bit or mixed-precision quantisation, pruning and layer fusion.

NPU Software Stack Weaknesses

These include a steep learning curve and the fact that SDKs and compilers are often vendor specific (limiting portability). Also, debugging and profiling tools are not as mature as they are for CPUs and GPUs. The issue to really watch out for though is that (bizarrely) not all neural network operations are supported by NPUs - and CPUs often have to pick on tasks like dynamic flow control and complex tensor operations, for instance. Accordingly, when looking for models to run on an NPU, and the intended operations, it is important to confirm which layers are fully accelerated and which revert to CPU execution. Workloads that require lots of back and forth between NPU and CPU will compromise performance, particularly real time.

NPU Programming Languages

Language	Pros	Cons
Python	Easy to use, high-level and well-supported.	Not used directly on-device (converted to lower-level formats).
C/C++	Fast and close to hardware.	Complex memory management and harder to debug.
Embedded C	Minimal footprint. Tight control .	Very low-level and error prone.
OpenCL / CUDA	High performance, parallelism.	Complex, not universal across all NPUs.
Vendor-specific DSLs / APIs (e.g. Hexagon NN API [Qualcomm])	Direct control over NPU execution.	Non-portable across devices. Steep learning curve and limited community support.

Example Frameworks for Vision-Based AI at the Edge on NPUs

Framework	Target NPUs	Pros	Cons
TensorFlow Lite	Many (e.g., Coral Edge TPU, Android NNAPI).	Lightweight, TFLite models can be accelerated via NNAPI or vendor delegates.	Limited flexibility, and conversion is required.
ONNX Runtime	Qualcomm SNPE, Intel OpenVINO, Rockchip, etc.	Interoperable with many frameworks as ONNX exports from PyTorch/ TensorFlow.	NPU support often via custom backends.
OpenVINO	Intel Myriad X.	Optimised for Intel hardware and there is a good computer vision toolchain.	Intel-specific.
SNPE (Qualcomm)	Hexagon DSP + NPU (Snapdragon).	Tight integration with Qualcomm chipsets and efficient.	Proprietary with limited documentation.
MediaTek NeuroPilot	MediaTek NPUs.	Integrates with Android NNAPI and TensorFlow Lite.	Android-focused and there is limited documentation publicly available.
NVIDIA TensorRT (Jetson)	NVIDIA NPUs (DLA), GPUs.	Extremely optimised. Deep TensorFlow/PyTorch support.	Not usable on non-NVIDIA hardware.

Return to Index ^^

CPU

Despite the rise of dedicated ICs that are geared for edge AI and especially vision-based applications, CPUs remain very popular. They are available as standalone devices (e.g. Intel Core i7, AMD EPYC) and are embedded into MPUs, MCUs and system-on-chip (SoC) devices such as Apple M1 and Raspberry Pi BCM2711.

CPUs handle general-purpose tasks well – offering a low/medium level of inference and pre-/post-processing capabilities - and they are flexible with good support for frameworks, libraries and languages. Not surprisingly, they have a very mature ecosystem with good compiler support, toolchains, debugging tools, SDK availability and OS-level support (e.g., Linux and RTOS).

However, CPUs (even multicore devices) have limited parallelism compared to GPUs and FPGAs, which limits throughput for deep learning inference. CPUs can also suffer from latency issues due to non-deterministic scheduling, even if they are multicore devices. Real-time kernels should be considered for time-critical inference.

CPU Software Stack Strengths

- Wide software support. Most AI/ML frameworks support CPU backends (TensorFlow, PyTorch, OpenCV, ONNX, etc.).
- Rich OS-level services. You can run full Linux distributions with networking, file systems, security, etc.
- Optimisation toolchains. Compilers like LLVM, GCC and AI accelerators like OpenVINO (Intel) or ARM Compute Library exist to optimise inference.

CPU Software Stack Weaknesses

- Less optimised for AI. Many AI frameworks prioritise GPU/NPU backends. CPU support is improving, but still slower.
- Software bloat. Full OS stacks can be heavyweight, which is not ideal for low-latency, real-time use unless carefully trimmed.
- Real-time constraints. Vanilla CPUs with general operating systems (e.g., Linux) aren't real-time unless customised (e.g., using PREEMPT_RT patches).

CPU Programming Languages

Language	Pros	Cons
C/C++	High performance, close to hardware and widely supported.	Complex memory management, slower development (potentially with lots of bug hunting).
Python	Easy to read/write. Fast development. Huge ML ecosystem (TensorFlow, PyTorch).	Slower execution. May need bindings (e.g., with C++) for performance.
Rust	Memory safety without garbage collection. Good performance. Increasingly popular for embedded.	Steep learning curve. A small but growing ecosystem.
Assembly	Max control and efficiency.	Extremely low-level. Rarely used unless optimising certain critical paths.

Example Frameworks for Vision-Based AI at the Edge on CPUs

Framework	Language	Pros	Cons
OpenCV (Vision processing library)	C++ (bindings for Python, Java, etc.)	Excellent for image pre/post-processing. Widely supported and lightweight. Integrates with DNN modules.	Direct DNN support is basic (compared to PyTorch/TensorFlow). Performance depends heavily on hardware optimisation.
ONNX Runtime (Inference engine for ONNX models)	C++, Python, C#	Lightweight and portable across hardware. Optimised CPU backends. Supports quantised models.	No training support (inference-only). Requires conversion from PyTorch/TF to ONNX.
TensorFlow Lite (Lightweight ML framework)	C++, Python	Optimised for mobile/edge. Good CPU performance with quantisation.	Conversion from full TensorFlow model can be tricky. Less transparent debugging.
PyTorch Mobile / TorchScript (ML frameworks)	C++, Python	Easier for developers already using PyTorch. Scripted models can run on CPU.	Less optimised than TFLite for small CPUs. Python dependency unless fully scripted.
Intel OpenVINO Optimised inference toolkit (Intel CPUs/VPUs)	C++, Python	Highly optimised for Intel CPUs. Post-training quantisation, model optimisation. Supports OpenCV integration.	Intel only, for best performance. Steep learning curve.
ARM Compute Library (Low-level optimised routines for ARM CPUs)	C++	High performance on ARM-based devices (e.g., Raspberry Pi). Optimised convolution and maths operations.	No high-level API (just building blocks). Steeper development effort.

Return to Index ^^

DSP

These are also commonly used in vision-based AI at the edge applications. Their strengths include low latency (making them suitable for real-time vision tasks) and their on-chip memory and parallelism.

They are optimised for specific operations (such as convolution and FFT), making them very power efficient and, as is implicit in the name, signal processing: because they have native instruction sets for matrix and vector operations. This last aspect makes them good at filtering, image enhancement, feature extraction and other key vision tasks.

DSPs are less general purpose than CPUs and GPUs, and deliver lower peak performance than the latter, though that might only be an issue if the application has massively parallel workloads (such as training a deep network). Also, DSPs in vision tasks depend on low-overhead transfer of data between accelerators. Unless memory bandwidth and DMA are correctly configured, bottlenecks might occur for some operations. Not surprisingly, as something of a specialist device, the DSP developer ecosystem is smaller.

DSP Software Stack Strengths

- Highly Optimised Libraries. Vendors like Qualcomm, TI and Cadence provide optimised libraries (e.g., Hexagon NN, TI Deep Learning [TIDL] and HiFi DSP SDK).
- RTOS Integration. DSPs are often used with an RTOS, making them ideal for deterministic applications.
- Tight Integration with SoCs. DSPs are often embedded – along with CPUs, NPUs and image signal processors (ISPs) - in heterogeneous SoCs, making cross-processing much easier via vendor SDKs.

DSP Software Stack Weaknesses

- Proprietary Toolchains. Many DSPs require vendor-specific compilers and toolchains (e.g., Qualcomm’s Hexagon SDK, TI Code Composer Studio), which can be limiting.
- Limited Framework Compatibility. TensorFlow Lite and ONNX often need custom conversion paths to run on DSPs.
- Manual Optimisation. Developers sometimes need to hand-optimise key routines using DSP intrinsics or assembly.
- Debugging is Harder. Debugging and profiling tools are less advanced compared to those for CPUs and GPUs, for instance.

DSP Programming Languages

Language	Pros	Cons
C/C++	Widely supported, low-level control and optimised libraries are available.	Manual memory management. Harder to debug.
Python (via conversion)	Used for model development. Compatible with TensorFlow Lite or ONNX	Not used directly on the device. Needs conversion to C/C++ or vendor intermediate representation.
Assembly (DSP-specific)	Maximum performance. Fine-grained control.	Tedious and error prone. Not portable.

Example Frameworks for Vision-Based AI at the Edge on DSPs

Framework	Vendor	Pros	Cons
Hexagon NN / SNPE	Qualcomm	Optimised for AI on Hexagon DSPs. Supports TFLite and ONNX models.	Proprietary. Limited flexibility.
TIDL (TI Deep Learning Library)	Texas Instruments	Supports vision models. Integrated with TI SoCs.	Complex build setup. Limited model support.
HiFi DSP SDK	Cadence	Audio and vision optimised. Good for low-power apps.	Niche use cases. Requires licensing.
TensorFlow Lite Micro	Various	Open source. Can be ported to DSPs.	Needs custom kernels. Limited performance without tuning.
ONNX Runtime (custom backends)	Various	Interoperable format. Supports conversion pipelines.	Backend tuning is required. Not always plug-and-play.

Return to Index ^^

FPGA

As mentioned, GPUs can perform thousands of operations in parallel and are low latency (sufficiently low for real-time). FPGAs, which have configurable logic blocks (see figure 4), tick these boxes too and are ideal for vision tasks such as object detection, classification and segmentation (all of which often involve parallel operations on pixels or regions).

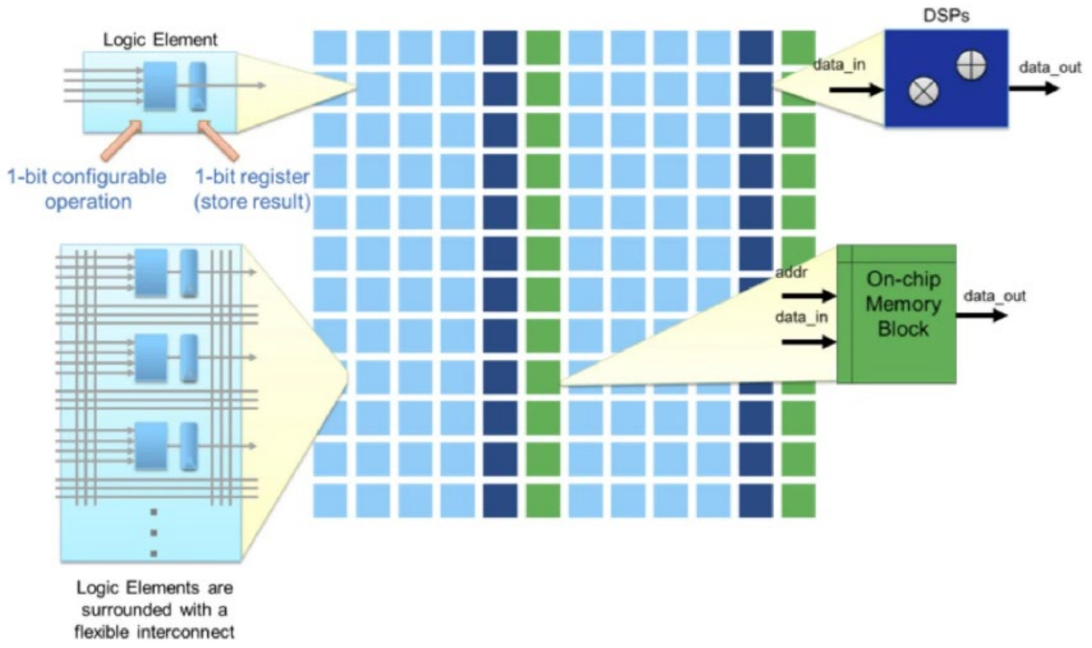
Logic Element
1-bit configurable operation
1-bit register (store result)
Logic Elements are surrounded with a flexible interconnect
DSPs
data_in
data_out
addr
data_in
On-chip Memory Block
data_out

Figure 4. FPGAs contain configurable logic elements. Some also contain DSP blocks, useful for dot-product calculations (a fundamental operation in linear algebra that is widely used in areas like ML and computer graphics). Source Edge AI + Vision Alliance. https://www.edge-ai-vision.com/2016/08/fpgas-for-deep-learning-based-vision-processing/

Unlike GPUs, which may have scheduling delays, FPGAs can not only deliver real-time performance, but they are also deterministic, making them perfect for safety-critical applications. FPGAs can be very energy efficient for certain workloads because they don't carry general computing overhead. Specifically, the internal hardware (logic gates and look up tables, LUTs) can be configured for the dataflow of a neural network model, including optimised pipelines, quantisation and even pruned models. However, depending on the application, long-term maintainability may need to be factored in, and may rule out the use of an FPGA. Specifically, whilst performance is deterministic, FPGAs might not be the best solution if the application is to have frequent model updates.

FPGA Software Stack Strengths

- When used alongside a traditional processor (though SoC FPGAs contain a CPU core and even an AI engine in the case of AMD's Versal ACAP), hardware-software co-design enables you to optimise the system architecture for performance and power. Essentially, you decide which tasks are best performed in software and which should be accelerated in hardware.

- Though FPGAs have been around since the 1980s, the popularity of AI is fuelling growing tool support. For example, vendors like AMD and Intel offer high-level synthesis (HLS) tools and AI-specific SDKs.

- Support for quantised models. Many FPGA tools work well with INT8 and lower precision models, reducing memory and power usage.

FPGA Software Stack Weaknesses

- There is a potentially steep learning curve. Programming an FPGA traditionally requires in-depth knowledge of hardware design languages (HDLs).
- FPGA-based AI development takes longer than Python frameworks, though linting tools improve code quality and portability.
- FPGA ecosystems are fragmented, with proprietary vendor tools limiting portability, though third-party tools enable faster, independent simulation and verification.
- Limited support for dynamic models. Unlike CPUs/GPUs, FPGAs struggle with models that change architecture dynamically or require runtime flexibility.

FPGA Programming Languages

Language	Description	Pros	Cons
VHDL / Verilog	Low-level Hardware Description Languages (HDLs).	Full control. Efficient/ mature toolchain.	Steep learning curve, verbose and error prone.
SystemVerilog	Modern HDL with object-oriented features.	Better modularity than Verilog/VHDL.	Still requires deep hardware knowledge.
High-Level Synthesis (HLS) (e.g., C/C++, OpenCL)	C/C++ code compiled to hardware logic.	Faster development. Easier for software engineers.	Often less efficient than handcrafted HDL.
Python (via tools like PYNQ)	Python wrapper for FPGA APIs (mainly for Xilinx boards).	Easy prototyping, accessible.	Not for low-level hardware development.

Example Frameworks for Vision-Based AI at the Edge on FPGAs

Framework	Languages	Pros	Cons
OpenCL for FPGAs (multiple vendors)	OpenCL C (based on C99 subset), C/C++ with OpenCL API	Cross-platform (with some caveats) and high-level programming for parallelism.	Performance varies. Long compile times. Less control than HDL.
Xilinx Vitis AI (for Xilinx FPGAs such as Zynq and Versal)	Python (for API), C++, HLS, VHDL	Pre-optimised Deep-Learning Processing Unit (DPU) for CNNs. Powerful profiling tools.	Steep learning curve for full toolchain. Tied to Xilinx hardware.
Intel OpenVINO + FPGA Plugin (for Intel FPGAs such as Arria and Stratix)	C++, OpenCL	Runs OpenVINO models on FPGAs. Pre-compiled bitstreams for some networks. Easy deployment from trained models.	Less customisation of hardware logic.

Return to Index ^^

MPU

Microprocessors are commonly used for vision-based AI at the edge due to their balance of computational capability, flexibility and software support.

Relative to their cost, they offer great performance (especially devices with 32- or 64-bit cores such as Arm Cortex-A) compared to CPUs. They also support Linux-based OSes (e.g., Yocto Linux and Ubuntu Core), which enable complex software stacks, computer vision libraries, and frameworks like OpenCV and TensorFlow Lite. In addition, many modern MPUs integrate AI/ML accelerators or GPU/ISP blocks to handle intensive computer vision workloads.

Understandably, as such a popular embedded system device, MPUs typically have interfaces for camera inputs (MIPI-CSI) that are crucial for vision systems. And many MPUs have real-time capabilities for latency-sensitive vision tasks. However, when several cameras are connected, the MPU's internal bus and memory architecture needs to be able to sustain the combined throughput without frame drops or inference lag.

MPU Software Stack Strengths

- Rich OS environment: Linux on MPUs enables multitasking, multi-threading, containerisation (e.g., Docker), and access to well-established software ecosystems.
- Broad AI framework support: TensorFlow Lite, ONNX Runtime, PyTorch (limited), and OpenCV are natively supported or easily cross-compiled.
- Custom ML model deployment: MPUs often support tools for quantisation, pruning, and cross-compilation of neural networks to run efficiently on-device.
- Good community and vendor support, especially for the most popular MPU devices.

MPU Software Stack Weaknesses

- Complex development: Building software for MPUs involves cross-compiling and managing dependencies.
- Latency and power: While better than cloud, MPUs are not always optimal for real-time, ultra-low-latency vision (a GPU or FPGAs might be better).
- Software fragmentation: Different vendors have different SDKs and toolchains (e.g., NXP's eIQ and TI's Edge AI SDK), which can cause portability issues.
- Security patching and updates: If the OS is embedded Linux, keeping it secure and up-t-date is non-trivial, especially in long-lifecycle devices.

MPU Programming Languages

Language	Use case	Pros	Cons
C/C++	Drivers, real-time components, OpenCV, GStreamer.	Fast, good hardware control, widespread.	Error prone. Not necessarily the safest language to use.
Python	Rapid prototyping, AI frameworks (TensorFlow Lite, PyTorch), OpenCV scripting.	Easy syntax, strong AI ecosystem.	Slower and needs Python runtime.
Shell scripts (Bash)	System-level automation, startup scripts.	Lightweight, integrated into Linux.	Not suitable for complex logic
Rust	Safe systems programming.	Memory safety, performance.	Smaller ecosystem, learning curve.

Example Frameworks for Vision-Based AI at the Edge on MPUs

Framework	Use case	Pros	Cons
OpenCV	Computer vision and image processing.	Open source and well-documented.	Can be heavy on the MPU's CPU.
TensorFlow Lite	ML inference.	Optimised for edge, supports quantisation.	Limited model support compared to full TensorFlow.
ONNX Runtime	Inference with various backends.	Interoperability across frameworks.	Less optimised for all hardware.
GStreamer	Video pipeline management.	Efficient streaming, integration with OpenCV.	Complex to configure.
Vendor SDKs (e.g., NXP eIQ, TI EdgeAI)	HW-specific acceleration.	Uses built-in accelerators, optimised.	Vendor lock-in. Steep learning curve.

Return to Index ^^

MCU

Microcontrollers are increasingly being used for vision-based AI at the edge due to their low power consumption, small footprint, and increasingly capable hardware (which offers real-time responses).

However, they have limited compute power - so the inference must be highly optimised – and limited RAM and Flash memory, limiting model size, input resolution and image buffers. MCUs can also suffer limited I/O bandwidth.

Despite these limitations, MCUs are still capable of being used in basic vision-based AI at the edge applications such as object recognition, gesture recognition, bar code / QR code reading, and industrial monitoring (e.g. defect detection).

Note:
MCU-based systems should have lightweight update mechanisms and memory-efficient model deployment to allow field maintenance.

MCU Software Stack Strengths

- Deterministic behaviour thanks to minimal software layers.
- Using an RTOS or going bare metal gives fine-grained control over scheduling and power management.

MCU Software Stack Weaknesses

- This is low-level development work that requires in-depth knowledge of hardware (e.g. register-level programming, direct memory access [DMA] and interrupts).
- The stack needs to be optimised by hand for core functions such as single instruction, multiple data (SIMD).
- Limited framework support, compared to Linux or Android environments, and therefore less compatibility with mainstream AI/ML frameworks.

MCU Programming Languages

Language	Pros	Cons
C	Extremely efficient (in terms of performance versus memory), ubiquitous in embedded environments, full control of memory and hardware.	Verbose and error prone (memory safety, buffer overflows). Poor abstraction for complex AI logic.
C++	Adds object oriented, templates, and some abstraction over C. Still highly efficient.	Feature-heavy (template metaprogramming, RTTI) can bloat code if not used carefully. Limited library ecosystem compared to desktop/server C++.
MicroPython / CircuitPython	Much easier and faster for prototyping.	High memory overhead. Very limited support for AI inference on vision data.
Rust	Memory safety without garbage collection. High performance with modern tooling.	Steep learning curve. Toolchain and ecosystem for embedded are still maturing.

Example Frameworks for Vision-Based AI at the Edge on MCUs

Language	Pros	
TensorFlow Lite for Microcontrollers (TFLM)	Designed specifically for MCUs (no dynamic memory allocation), good support for quantised models (INT8) and good community support.	Limited operator support (e.g., no support for large or complex layers), it lacks model training (must use pre-trained models from TensorFlow), and model conversion and optimisation can be tricky.
CMSIS-NN (Arm Cortex-M CPUs)	Highly optimised NN kernels for Arm Cortex-M. Works well with TFLM for efficient inference.	Only provides kernels (no model compiler or graph representation), low-level and harder to use alone.
Edge Impulse	Web-based platform for data collection, training, and deployment to MCUs. Supports TFLM backend. Great for rapid prototyping and deployment	Less flexible for custom models. Requires cloud platform for training. Not ideal for full control or proprietary pipelines.
NNoM	Lightweight NN inference library for MCUs. Fully written in C.	Not widely adopted. Less community support and documentation.

Return to Index ^^

TPU

Tensor Processing Units are specialised hardware accelerators designed by Google primarily for accelerating machine learning workloads, particularly those involving neural networks. In the context of vision-based AI at the edge, TPUs are increasingly used because they offer a unique mix of power efficiency, speed, and parallelism: which is critical when deploying AI models in edge devices like cameras, drones, smartphones and IoT systems.

TPUs are optimised for the kinds of matrix operations that dominate vision tasks (e.g., convolutions in CNNs). Also, dedicated edge TPUs (e.g., Google's Coral TPU) are specifically designed for low-latency inference on small, efficient models like MobileNet.

They can be embedded into tiny systems (e.g., Coral USB Accelerator) - making them ideal for compact edge devices - and are particularly fast when working with 8-bit quantised models, which are common in edge deployments to reduce memory and improve inference speed.

TPU Software Stack Strengths

Edge TPUs are tightly integrated with TensorFlow Lite, Google's lightweight framework for mobile and embedded devices, and optimised TFLite models can be compiled directly using the Edge TPU Compiler. Full support and documentation from Google make the development process smoother if you stay within their toolchain, and integration with other Google tools (e.g., Colab and Cloud AI) is seamless.

TPU Software Stack Weaknesses

Because TPUs are optimised for specific workloads and operations, custom layers or complex architectures not supported (by the TPU) will need to be performed by a CPU or MPU, for example. Edge TPUs only support a subset of TensorFlow operations. Also, only quantised (INT8) models are supported, so additional steps like post-training quantisation or QAT, which can be complex, are required. Precision loss from quantisation can degrade accuracy if not handled carefully. Understandably, the software stack (Edge TPU Compiler, runtime, etc.) is tightly controlled by Google, and there is less community-driven support compared to more open platforms (e.g., NVIDIA Jetson with PyTorch).

TPU Programming Languages

Language	Usage	Pros	Cons
Python	Main language for model development and deployment.	Easy to use. Supported by TensorFlow, TFLite, and Edge TPU runtime.	Less efficient for low-level operations.
C++	Used for custom applications using TFLite C++ APIs.	High performance, low overhead.	Steeper learning curve; more boilerplate.
Shell (CLI)	For compiling models with Edge TPU Compiler.	Quick integration into deployment pipeline.	Minimal logic possible.

Example Frameworks for Vision-Based AI at the Edge on TPUs

Framework	Usage	Pros	Cons
TensorFlow + TensorFlow Lite	Primary framework for training and deploying models to TPUs.	Excellent support (end-to-end flow) with good tools for quantization and model optimisation.	Steep learning curve.
PyTorch (via ONNX → TFLite)	Possible via conversion to TFLite.	Familiar syntax and a strong community.	Conversion to TFLite may break some operations, and quantisation not as mature.
Edge TPU Runtime	Required to execute models on Edge TPU.	Fast and optimised.	Supports only a limited set of models and operations.

Return to Index ^^

Vision-Ready SOMs

Having discussed the above core hardware technologies let's consider commercially available vision system on modules (vision-SOMs) that lend themselves well to developing AI-enabled applications. Figure 5 shows an example.

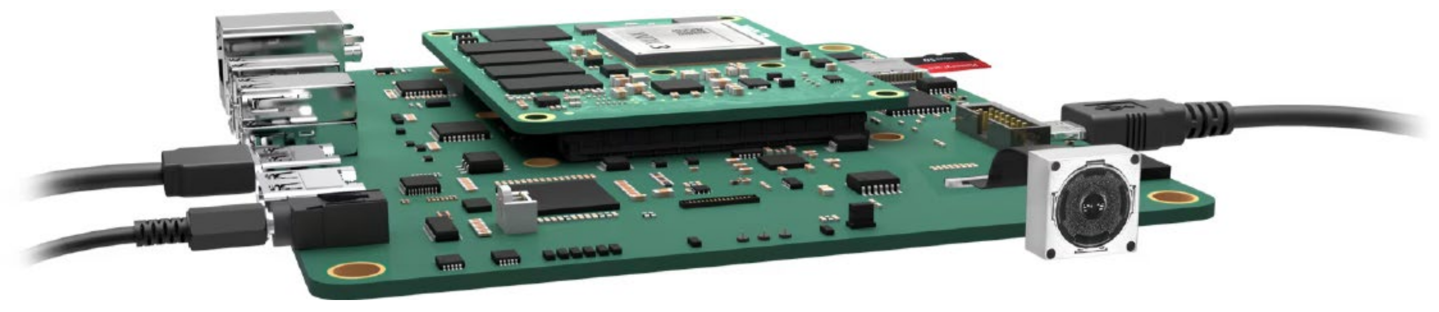

Figure 5. Machine vision requires embedded systems that can analyse data on the spot and offer configurable sensor capabilities. SOMs enable developers to take advantage of machine vision at scale while keeping costs low.
Source AMD: https://www.amd.com/en/products/system-on-modules/what-is-a-som.html

The advantages of using a vision-ready SOM include:

- Lower risk. Developing custom hardware comes with significant risk such as hardware bugs, power and thermal management issues and potential manufacturing defects. Also, the extent of its compatibility with other systems might not be known until it is in the field. SOMs on the other hand are field proven, tested for stability and often supported by their OEMs or third parties.

- Reduced development time. SOMs come pre-designed with a working processing 'engine' (be it a GPU, MPU, DSP etc.), memory (RAM and ROM) and I/O interfaces. This removes the need to design and test complex hardware subsystems, allowing the project to focus on software development and system integration. Vision-SOMs provide even more - specifically dedicated pre-integrated camera interfaces (such as MIPI CSI or parallel interfaces) - reducing design effort even further.

- Reduced certification burden. SOMs often carry regulatory pre-certifications (e.g., FCC, CE), reducing the burden on your own product certification process. This is especially advantageous for wireless-enabled modules or medical/industrial systems.

- Ecosystem and community support. Popular SOMs have active communities and support channels, which speeds up troubleshooting and provides access to tutorials, libraries and example code.

- Integrated AI acceleration for real-time inference. Many vision-SOMs (e.g., NVIDIA Jetson, Google Coral, NXP i.MX 8M Plus SOMs) include AI/ML hardware accelerators for the core processing engine (again, GPU, NPU or DSP, for example).

- Stack availability. Vision SOMs typically include or support Linux distributions (Yocto, Ubuntu, etc.), AI SDKs (TensorRT, OpenCV, GStreamer, etc.) and pre-built drivers for vision peripherals. This dramatically lowers software integration effort, especially for camera and sensor support.

- Modularity and scalability. SOMs can be easily upgraded (e.g., moving from a Jetson Nano to a Jetson Xavier) while keeping the same baseboard. This allows for products to be scaled/upgraded with minimal redesign effort but be mindful of starting off with too little or too much scope for expansion. Also, you will of course be tied to the SOM's physical spec (including interfaces).
- Cost-effective for low- to mid-volumes. Vision-SOMs are very cost-effective at prototype and production volumes under 10k units, where NRE costs of a custom PCB would be prohibitive.

A word of warning: SOM availability changes fast as, understandably, new products are being launched to serve the growing market that is AI at the edge. Accordingly, developers must check module lifecycles before anything is finalised.

Also, be mindful of the fact that SOM manufacturers vary massively in SDK and OS update schedules. If the system you are developing will be deployed for a few years you need to be confident that there will be long-term support for kernels and drivers. And it is recommended that your organisation/team sets reminders to check if (or rather when) support will end.

Vison-Ready AI Computing Platform: An Example

A step beyond using a vison-ready SOM is to use an industrial / edge AI platform. The main advantage of doing so is that you start your project with a turnkey solution: a validated hardware / software environment that drastically shortens development time, improves reliability and performance for real-time vision at the edge. Integration risk is lowered too.

So, what might a platform comprise? To best answer that, let's consider an example: Innodisk's APEX-X100 (see Figure 6). It is offered with an NVIDIA RTX 6000 Ada accelerator (18,176 CUDA cores, 568 Tensor cores, 142 RT cores; 48GB GDDR6) which gives large inference/fine-tuning headroom for modern vision models and multimodal workloads. That's useful when you need high throughput (multi-camera) or want to run larger models locally.

The APEX-X100 also comes with Intel 13th-Gen Core i7 or i9 options, up to 128GB (or more in some variants) of DDR5 and 512GB–1TB (or higher) NVMe pre-installed, all of which is ideal for preprocessing, batching and running auxiliary services.

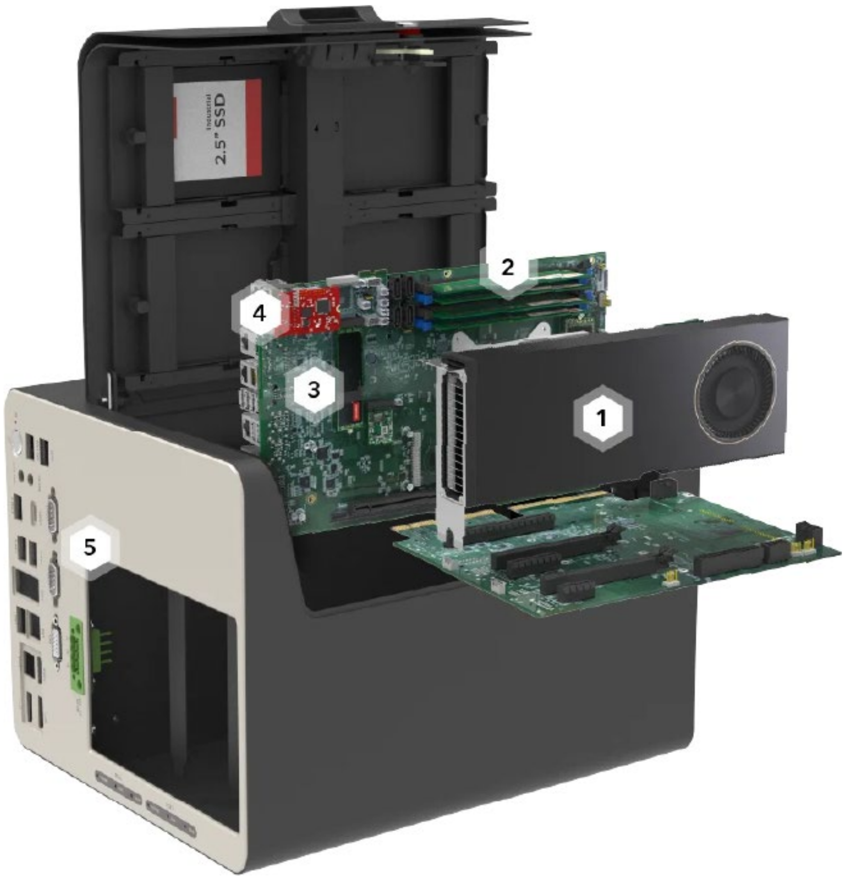

Figure 6. The APEX-X100 is an industrial / edge AI computing platform designed to support demanding vision and compute heavy workloads such as local model training, inference and fine tuning. 1 = the compute hardware. 2 = a DRAM module that supports up to four Innodisk DDR5 4400 UDIMM modules. 3 = Flash storage (featuring an Innodisk industrial-grade M.2 4TG2-P 512GB SSD with a PCIe Gen 4 x4 interface). 4 =an out-of-band remote management module. 5 = I/O ports (specifically three 2.5Gbps LANs, a 10Gbps LAN, multiple high speed USB ports and various COM/DIO ports).
Source Innodisk. https://www.innodisk.com/en/edge-ai-systems/nvidia-solution/apex-x100

In terms of how the APEX-X100 might be used by software developers:

Model Selection, Deployment and Optimisation

- Models can be selected that take advantage of the GPU /Tensor cores. Example models include YOLO (v7/v8), Faster R CNN and Mask R CNN.
- Use frameworks and inference acceleration libraries such as NVIDIA's TensorRT, ONNX Runtime with CUDA, cuDNN; possibly even model quantisation / pruning for faster inference.
- For fine tuning / training, you can ensure the model fits within the GPU memory.

Model Selection, Deployment and Optimisation

- Camera ingress. You'll need to connect cameras / vision sensors. The APEX X100 offers many I/O options (USB, 10/2.5 GbE, etc). If you use MIPI or GMSL cameras, you might need additional capture hardware or interface boards.
- Storage and data pipeline. Real time vision yields large amounts of data. Fast NVMe for buffering or storing images/videos. Possibly RAID or external storage for archiving. Tip: use SSDs with industrial grade durability for reliability.
- Networking / communication. Streaming video or sending inference results over a network requires high bandwidth (10 Gbps or multiple 2.5 Gbps links) and reliable network stack. Edge deployment may require remote management and monitoring (the APEX X100's OOB helps in this respect).

Model Selection, Deployment and Optimisation

- OS / runtime environment. Your application could run on Windows IoT / Windows Server but many vision/AI frameworks run better on Linux (e.g. PyTorch, TensorFlow, OpenCV, etc.) or containers (which better support portability).
- Inference / training pipeline. This is from data ingestion (from cameras) through to pre-processing (resizing, normalisation), batch or streaming inference, post processing, and visualisation and/or alerting.

> **Tip**
> *You should use the exact same number and types of cameras that will be used in the field, as the use of multiple high bandwidth cameras will directly impact disk I/O and buffering.*

- Edge/cloud syncing. If data logging or model retraining happens centrally, you may need to sync data/models with the cloud or central servers. APEX X100 could act as a node in a larger distributed system.
- Security and maintenance. You need secure boot, OS updates, encrypted storage and be sure to safe firmware/drivers. The remote / OOB management will help with maintaining and monitoring health (especially if deployed in remote / harsh environments). Note: the OOB management should be integrated into software workflows for monitoring and recovery, and not just hardware-level control. Indeed, hardware-level control might not even be possible for some devices that communicate over wired protocols such as Ethernet.

Return to Index ^^

Cameras

When selecting a camera for a vision-based AI at the edge application, there are several critical factors to consider, and decisions made early on in the flow will have direct implications on ease of integration, system performance, power consumption and, ultimately, the success of the project.

Making those decisions is a system-level one and software engineers must be included to ensure the chosen camera does not become a bottleneck during development or deployment. Also, in the list that follows, not all points raised are of direct relevance to the project's software engineers, but we share them on the premise that there is no harm in raising awareness and presenting a broader picture.

What to look for and why

- Resolution. Higher resolution means more detail but also more data to process, so choose a resolution that matches the needs of your AI model (e.g., object detection may need less detail than facial recognition).
- Frame Rate (FPS). Real-time applications (e.g., robotics, surveillance) may require 30 FPS or higher whereas slower frame rates might suffice for static or periodic tasks (e.g., industrial inspection).
- Frame Synchronisation and Spatial Calibration. These are vital for accurate detection/tracking in multi-camera system.
- Interface and Connectivity. Options include USB, MIPI CSI and Ethernet - but the choice may be driven by the edge hardware - and you should consider bandwidth and driver support.
- Sensor Type. CMOS is the most common type as it is faster and more power-efficient than CCD. As for colour (RGB) vs monochrome, the latter performs better in low light and offers higher contrast for certain AI tasks.
- Lens and Field of View (FoV). Understandably, wide-angle is best for monitoring or tracking and narrow FoV is better for detail or distance work. Subject variability will govern whether you opt for fixed- or auto-focus. Note: Camera housings and mounting should also be evaluated for the deployment environment.
- Lighting Conditions. These will drive the camera's requirements in terms of low-light sensitivity, IR capability (for night vision) or HDR support for high-contrast scenes. Also, some edge cameras support global shutter for fast-moving objects.
- Driver and Software Support. Check the availability of drivers for your edge platform (Linux, Android, etc.) as well as support for GStreamer, OpenCV and AI SDKs (NVIDIA Jetson, OpenVINO, etc.), for example.

Camera selection

The inclusion of software engineers in camera selection is a must. It ensures the selected camera interfaces properly with the platform, AI frameworks (e.g., TensorFlow Lite, PyTorch), and drivers. Also, the camera's output directly affects inference performance and memory usage, and when it comes to prototyping it will be the software engineers that will be largely responsible for investigating any frame drops, latency issues or poor image quality.

In terms of ensuring maintainability, there is a strong argument for standardised camera modules (an example of which is shown in Figure 7).

Figure 7. Above, Innodisk's EV8U-LSM-RLCF is a USB 2.0 8MP resolution, 30fps fixed focus camera module with OS support for Windows, Linux and Android. It can be used in low light conditions thanks to an integrated image signal processor (ISP)
Source: https://www.innodisk.com/en/products/camera/usb-20/ev8u-lsm1-rlcf

Return to Index ^^

Customisation

Whilst we extolled the benefits of going down the SOM or vision-ready AI platform route, customisation almost always becomes necessary to align the hardware, software, and deployment environment with the specific requirements of your use case. Most if not all of the customisation you will be doing yourself, so please note the following...

Pitfalls to avoid:

- Underestimating Integration Complexity. Hardware and software stacks (e.g., camera SDKs + inference runtime) can conflict. Also, while early proof-of-concept systems may work fine, scaling to production often exposes bottlenecks (which is why, above, we recommend using end application cameras in their intended quantities).

- Thermal or Power Issues. Many teams forget to test under sustained load, causing throttling or brownouts.

- Portability. Choosing accelerators or SDKs with limited cross-platform support can trap you. Ensure your models and code are portable.

- Insufficient Lifecycle Planning. Industrial deployments often need 5–10 years of component availability and sectors such as medical, possibly longer. Pick components (storage, GPU, NIC) with known lifecycle commitments. Note, Innodisk has an excellent reputation in this respect, and it certainly an area in which Simms offers value (see later).

- Overlooking Maintenance and Updates. Again, we have already mentioned that the first day of deployment is just the start on the vision-based, AI-enabled system's life at the edge. Devices need secure, remote OTA updates. Overlooking this will almost certainly lead to operational pain later.

- Poor Dataset Fit. Again, when we discussed the flow, we stressed that edge environments can differ in lighting, motion blur, or camera angles. You may need to use custom datasets and on-site fine-tuning to assure robust performance.

However, even if your team has good, all-round engineering and AI/ML skills, outsourcing some parts of the customisation often makes sense. Reasons include faster time to market; help with meeting regulatory and reliability compliance requirements; increasing your confidence in your ability to assure your customers that lifecycle management and long-term support are available; and help with ruggedising your system for life in a harsh environment (for example, SSD endurance through power-loss protection).

Where Innodisk adds value:

- Hardware and Firmware Co-Design. Deep control of SSD firmware, DRAM validation, and industrial motherboards. Innodisk can optimise I/O throughput and storage performance specifically for your AI vision workload. The OEM can also personalise your system's BIOs for unique usage and increase system efficiency.
- Vision-Ready Platform Experience (APEX Series). Already integrated with NVIDIA/Qualcomm accelerators, so custom thermal and power tuning is streamlined. Ready for industrial edge environments — rugged, compact, and stable.
- Turnkey Integration and Validation. Innodisk can test your AI model on their hardware, check compatibility, and ensure long-term stability.
- Lifecycle and Supply Chain Stability. As an industrial supplier, Innodisk manages long-term availability and batch consistency, unlike vendors largely focussed on supplying consumer hardware.

Return to Index ^^

Why Simms?

We can help accelerate your vision-based AI at the Edge project

In this white paper we have provided a wealth of information - not only offering advice but also warning of pitfalls – to help you fasttrack the development of your project. Further acceleration can be achieved through establishing good relationships with key suppliers, particularly distributors that not only provide good technical support themselves but also have direct access to the specialists within the OEM organisations they represent.

If you are developing a vision-based AI at the edge system around a core processing device (and we have discussed many types in this white paper) you will also need industrial-grade memory (RAM and NVM) and embedded peripherals, cameras as a minimum. You will probably need SDKs and prototyping boards too.

Alternatively, and as stressed above, vison-ready SOMs and AI computing platforms boast many benefits, not the least of which is a shorter time to market. It is essential to select the most appropriate SOM or platform for not only your immediate project but also your long-term objectives: for example, IP re-use on subsequent projects. And as emphasised throughout this paper, if the underlying hardware wrong (incorrectly selected), many software-controlled functions might be constrained, and overall performance compromised.

Simms can help you make the right choices and supply the hardware plus any support you need to accelerate your project. Indeed, the company brings deep technical expertise, helping you align your software architecture and performance goals with the optimal hardware configuration.

As a specialist distributor, Simms bridges the gap between developers, system architects and world-leading manufacturers such as Innodisk. The company connects software innovators (like you) who are building vision-based AI at the edge with the hardware platforms that make their vision a reality; thus helping them move as fast as possible from concept to deployment: with solutions that are reliable, scalable and ready for industrial environments.

Importantly, Simms works collaboratively and in a structured flow, that begins with developing a clear understanding of the application and workload. From there, Simms matches the intended software environment (including models) to the most suitable compute, memory, and storage technologies to accelerate development and de-risk deployment.

In essence, Simms is far more than a distributor. When you engage with the company you benefit from a sound bridge between intelligent software and industrial-grade hardware, backed by long-standing vendor relationships and decades of embedded experience.

Return to Index ^^

Our partner Innodisk

Selecting the right compute module is only one part of delivering a robust vision-AI edge system.

innodisk

Real-world deployments depend heavily on how well the surrounding hardware performs over time: the endurance of storage under constant data ingest, the stability of DRAM under sustained inference loads, the consistency of component behaviour across production batches and the ease with which devices can be maintained in the field. This is where Innodisk excels.

Industrial Memory for AI Workloads

Unlike consumer or enterprise SSD vendors, Innodisk engineers the firmware, NAND selection, and controller behaviour around the realities of edge vision systems. Models are cached and updated frequently; cameras continuously generate high-write workloads; local logs accumulate; inference pipelines generate random access patterns. Innodisk SSDs are optimised for these patterns and incorporate features such as power-loss protection, advanced wear-levelling and configurable over-provisioning, ensuring inference pipelines remain stable as systems age.

Validated DRAM for Sustained Compute

AI inference on GPUs, NPUs or accelerators often exposes weak points in memory design—temperature drift, timing errors, or unpredictable throttle events. Innodisk's industrial DRAM modules are validated in wide operating temperature ranges and under non-stop load conditions, reducing the "silent" instability issues that can derail edge AI projects and are notoriously difficult to diagnose at the application layer.

Vision-Ready Platforms, Not Just Components

Innodisk's APEX series represents a step beyond discrete memory and storage. These platforms give development teams a pre-validated environment where GPU acceleration, DDR5 capacity, PCIe storage, cooling, and remote management have already been engineered to work together. This avoids the integration trap many teams face - where individually excellent components underperform when put into a 24/7, multi-camera deployment.

Deployment, Recovery and Lifecycle

Edge AI systems often operate in locations that are inaccessible or mission critical. Innodisk's out-of-band management modules allow devices to be recovered, updated, or re-imaged without OS-level access. Combined with long-term component availability and strict batch-to-batch consistency, this gives engineering teams confidence that prototypes will behave the same as production hardware, and that deployed units will remain supportable years later.

Simms + Innodisk = Faster, Safer Deployment

Through our relationship with Innodisk, Simms can bring you into direct contact with their engineering teams. This includes firmware tuning for endurance profiles, pre-deployment validation of your AI workloads, platform-level customisation, and lifecycle planning. Rather than selecting memory or compute in isolation, we help you align the behaviour of your models, data ingestion pipeline and system architecture with the exact hardware that will support it for the duration of its life in the field.

Return to Index ^^

Summary

There is a growing demand for vision-based AI at the edge system in virtually every industry sector. In developing those systems, software engineers face many challenges that can be made all the more difficult by the underlying hardware.

Its strengths and weakness in relation to the software stack must be fully appreciated; and software engineers must be involved in hardware selection if the project is to run on time and risks reduced.

In terms of fast tracking (and further reducing risk) SOMs and AI-ready platforms offer a compelling abstraction layer between AI software and edge hardware, allowing software developers to focus on high-level optimisation. Indeed, by leveraging SOMs / AI-ready platforms with built-in support for neural accelerators, vision pipelines and robust SDKs, teams can prototype rapidly, scale efficiently and deploy confidently in the field.

Lastly, do not overlook the considerable role the distributor of the underlying hardware can play in accelerating your project. They can help with hardware selection, and they have direct contact with the OEMs, who in turn can assist and even customise solutions for you.

About the Authors

This white paper was written by Simms technical specialists.

Return to Index ^^

Useful Links

Edge AI + Vision Alliance.

A global industry partnership (100+ member companies) focused on edge AI + computer vision. Supports product creators, offers education, market/technology-trends insight, and addresses "bringing vision + AI to products" challenges.

- www.edge-ai-vision.com

EDGE AI Foundation.

A foundation dedicated to edge AI (including tinyML, embedded systems) covering hardware, software, ecosystems, and deployment. Good fit when discussing "AI-ready modules" and ecosystems.

- www.edgeaifoundation.org

SOM community and support resources:

NVIDIA Jetson (Nano, Xavier, Orin series):

NVIDIA Developer Forums (Jetson Subforum)

- forums.developer.nvidia.com/c/jetson-embedded-systems/70

JetsonHacks

- www.jetsonhacks.com

GitHub Projects

- www.github.com/dusty-nv/jetson-inference

Google Coral (Edge TPU SOMs):

Coral Community Forum

- www.coral.ai/community/

GitHub: Coral Dev Board / Edge TPU Examples

- www.github.com/google-coral

NXP i.MX 8M Plus SOMs (Toradex, Variscite, TechNexion, etc.):

NXP Community (MCU & i.MX forums)

- community.nxp.com

Toradex Developer Center

- developer.toradex.com

Variscite Forums and Wiki

- www.variwiki.com

TechNexion Community

- www.technexion.com

Raspberry Pi Compute Module (CM4/CM5) (often used for lightweight vision applications):

Raspberry Pi Forums

- forums.raspberrypi.com

Return to Index ^^

simms

Get in touch

01622 852800
www.simms.co.uk/intelligent-solutions
sales@simms.co.uk

Simms International
Northdown Close
Northdown Business Park
Ashford Road
Lenham
Kent
ME17 2DL