Proposed x147 Photonic Computing Architecture: Design Overview and Plan

Executive Stance

Don’t attempt an “optical x86” – instead, build a **new ISA (x147)** with native photonic acceleration and an optical fabric, while handling conventional control flow in CMOS. The strategy is to keep standard **control, branching, and memory operations in CMOS**, and use **photonic MAC islands** and an **optical interconnect** for high-throughput linear algebra. Maintain **x86/x64 compatibility** via hardware-assisted **binary translation** (JIT) rather than trying to run legacy code directly in optical logic. In short: **x147** is a clean-slate 64-bit ISA (with a photonic extension) that achieves compatibility by translation or virtualization – *not* by being an “optical clone” of x86.

Key new components include an **ejectable ROM** module (for secure boot and calibration data), a **“free-form” memory control panel** (to tune memory and interconnect bandwidth allocation in real time), a custom **x43 micro-hypervisor** kernel (with strict separation for photonic real-time loops vs general-purpose code), and tightly integrated **photonic computing tiles + optical fabric** (as first-class parts of the architecture, not bolt-on peripherals). The **ejectable ROM** should be physically keyed and cryptographically verified, serving as a root-of-trust and a carrier for photonic calibration data. The **free-form memory panel** provides real knobs for bandwidth arbitration (HBM vs. DDR memory, GPU vs CPU vs photonic accelerator priorities, optical fabric credits, etc.), exposing controls that *actually matter* (not just notional sliders).

Summary of stance: Build **x147** + **photonic compute** + **optical fabric** into a cohesive system. Use binary translation for legacy **x86** software (with hardware support and a caching translator) to handle backward compatibility. Make clear that ring-0 x86 code won’t run natively – it will run in a VM if needed. By delineating these boundaries, we avoid quixotic efforts (e.g., trying to recreate x86 in optics) and focus on where photonics truly excels (fast linear algebra and high-bandwidth I/O)cambridgeconsultants.comcambridgeconsultants.com.

System Architecture Overview

The **x147** SoC is a heterogeneous system combining a conventional high-performance CPU cluster, integrated photonic accelerator tiles, high-bandwidth memory, and an optical I/O fabric. A secure boot chain is implemented via on-board and removable ROM modules. A conceptual block diagram is shown below:

```
[ Ejectable ROM Bay ]

  |  SPI/eSPI + Auth

  v

[ Boot ROM (on-board) ] --> [ Root of Trust ] --> [ Secure Boot ]
```

```
                    |
                    v
+--------------------------------------------------------------------------------+
|                    x147 SoC (CMOS, 5–7 nm)                      |
|                                                              |
|  CPU Complex:              Mem & Coherence:           IO & Fabric:           |
|  - 8–32 x147 cores          - L2 per cluster           - PCIe 5/6, CXL 3.x      |
|  - 128-bit vector (VX)       - Shared L3                - Optical Fabric Ctrl   |
|  - OPTX ext (photonic ops)    - MESI-TSO (x86-friendly)    - 2× O-Fabric ports     |
|                                                              |
|  Photonic Compute Tile(s):                Memory Subsystem:                 |
|  - MZI mesh 64×64 oMACs                   - 4–8 HBM stacks (1–3 TB/s each)    |
|  - Modulators + PD arrays                 - 4–8 DDR5/6 channels              |
|  - On-tile tap PDs (H logging)             - SRAM scratch for oMAC batching    |
|  - OPTX DMA, 10 ns loop                   - “Bandwidth Arbiter” (programmable)|
|                                                              |
|  Graphics Cluster (GCU):                  Security / ROM:                  |
|  - Vector/TMU/Raster + OPTX path              - OTP fuses, PUF                  |
|  - Optical interposer to VRAM               - Ejectable ROM keys & capsules     |
+--------------------------------------------------------------------------------+
       |                            |
       |                            +-- [ Free-Form Memory Panel ]
       |                                (Partitions/Weights/Telemetry)
       v
  [ Optical Backplane / Photonic Fabric ]  <-->  Other CPUs/GPUs/Memory Bricks
```

Figure: System Block Diagram. The **x147 SoC** (central box) contains standard CPU cores, coherent caches, memory controllers, I/O including PCIe5/6 and CXL 3.0, plus controllers for the optical fabric. Integrated on the package are one or more **photonic compute tiles** (each a 64×64 Mach-Zehnder interferometer mesh for optical MAC operations), tightly connected to the SoC via silicon interposer. High-bandwidth memory (HBM) and DDR provide the memory hierarchy, and a **graphics compute unit (GCU)** is included for rasterization and GPU-like tasks, with an optical link to its VRAM. The **ejectable ROM bay** interfaces via SPI and is part of the secure boot chain and photonics calibration system. A **Free-Form Memory Panel** (exposed to users/administrators) allows dynamic control over memory and fabric bandwidth allocation policies.

This architecture is explicitly designed to **integrate photonics as a first-class computing resource** alongside traditional electronics. Photonic elements (compute tiles and fabric) are co-packaged and coherently interfaced, not treated as external add-ons. This follows industry trends where photonics is moving from just optical links toward **photonic interposers and co-packaged optics** deeply integrated in computing systemscambridgeconsultants.comcambridgeconsultants.com. The optical backplane can connect multiple sockets or accelerator bricks at terabit per second speeds, using photonic links in place of electrical backplanes for improved bandwidth and energy efficiencycambridgeconsultants.combusinesswire.com.

x147 Processing Unit (Custom ISA and Microarchitecture)

ISA Design: x147 is a 64-bit RISC-leaning ISA, designed for clean semantics and easy integration of photonic acceleration. Key features of the ISA:

- **128-bit vector** instructions (VX) as a baseline, with an option to scale to 256-bit by pairing operations (to keep decoder complexity low). This provides SIMD capabilities analogous to SSE/AVX on x86, but in a cleaner RISC-style package.
- **OPTX photonic extension:** a set of instructions and command queue semantics to dispatch optical matrix operations (matrix-vector multiply, FFT, convolution, etc.) to the photonic tiles. These include descriptor-based DMA commands to feed data to the photonic mesh, and fixed-latency fence instructions to synchronize with the <10 ns optical compute loops.
- A **Total Store Order (TSO) memory model**, matching x86's memory ordering. By making x147 natively TSO (with inexpensive fence instructions), it simplifies the correctness of dynamically translating x86 code, which assumes TSO. Aligning the memory model avoids the costly memory-order emulation that would be needed if the new ISA were weaker than x86 in ordering guarantees.
- Complete set of atomic operations (LL/SC as fundamental primitives, plus fetch-add, compare-and-swap, and masked atomics) that cover the needs of higher-level languages and match expected x86 atomic semantics. This ensures that x86 synchronization primitives can be mapped or implemented correctly on x147.
- Privilege levels: **M-mode** (machine firmware, e.g. for boot and low-level control), **S-mode** (supervisor, for the x43 kernel/hypervisor), **U-mode** (user applications). Additionally, an **H-mode** (hypervisor) is defined to support full virtualization of guest OSes when needed. This is similar to RISC-V's privilege model with an extra layer for virtualization.

Microarchitecture: The x147 CPU complex consists of **8–32 out-of-order cores**, each a moderate 3–4-wide dispatch design, targeting clock speeds of ~3 GHz or higher (in 5–7 nm CMOS technology). The design emphasizes a balanced pipeline that can exploit instruction-level parallelism without an extreme issue width (to keep power reasonable).

- Each cluster of 4 cores shares a L2 cache (approximately 1–2 MB per core or per cluster, configurable). A larger shared L3 cache of 16–64 MB serves all cores, maintaining coherence across the chip.
- **Cache coherence** uses a MESI protocol tuned for TSO (ensuring writes become visible in order), which makes it easier to maintain memory ordering equivalence with x86 when running translated code. The coherence and memory subsystem are designed to be **x86-friendly**, meaning they won't reorder memory operations in ways that violate x86 expectations, reducing the overhead on the translator to enforce ordering.
- The core includes a **μop cache** specifically in the translation engine. This acts as a cache for hot instruction sequences that have been translated from x86 into x147 μops. It functions similarly to how modern x86 implementations cache decoded μops for performanceblog.kowalczyk.infoblog.kowalczyk.info. In this system, however, it caches the result of the dynamic binary translation (DBT) of legacy code: frequently executed x86 instruction sequences get translated once to an optimized sequence of x147 operations and stored for reuse, greatly accelerating subsequent executions of the same code paths (much as Transmeta's Code Morphing software and modern Apple Rosetta 2 do).
- The **128-bit vector (VX)** units are integrated in each core, providing baseline data-parallel acceleration. These can handle typical SIMD workloads and also assist graphics (the GPU cluster can reuse these units when appropriate).

Graphics Cluster (GCU): A graphics processing cluster is included, but kept relatively straightforward. It provides basic rasterization, texture mapping, and shading capabilities (akin to a simplified GPU) primarily for display and visualization tasks. It likely includes specialized fixed-function units for texture sampling and geometry, but for compute it can leverage the same 128-bit vector units (VX) present in the CPU, as well as the photonic OPTX acceleration for suitable workloads. The GCU is connected to its video memory (VRAM) via an **optical interposer** or optical link, enabling high bandwidth between the GPU and its memory with low latency. (If packaging allows, the VRAM might even be unified on the HBM stacks instead of a separate memory, simplifying the architecture and potentially offering a unified memory model for CPU and GPU). Using an optical interconnect for the GPU's high-bandwidth memory access is in line with recent efforts to replace copper with photonics in GPU memory systemstspasemiconductor.substack.comspectrum.ieee.org, allowing extreme bandwidth scaling without the signal integrity and energy problems of copper at high speeds.

Why not make it an "x86 chip"? By designing a fresh ISA, x147, we avoid the baggage and complexity of the x86 architecture in the core. This yields a cleaner pipeline and more efficient execution for new code. The decision to **not run arbitrary x86 code in hardware** (especially not in kernel mode) means we can **sandbox legacy code** and use translation or virtualization as needed. This ensures that any quirky or unsafe legacy behavior (old 16-bit modes, self-modifying code, deprecated segmentation features, etc.) does not pollute the new design – those can be handled at a higher level in software translation where we have more flexibility. History has shown that dynamic binary translation can be quite effective: for example, Intel's IA-32 Execution Layer on Itanium achieved around ~60% of native performance for x86 programsblog.kowalczyk.infoblog.kowalczyk.info, and Transmeta's code morphing demonstrated that a well-designed translator with hardware support can approach native speeds in many casesblog.kowalczyk.info. Modern just-in-time translation (like Apple's Rosetta 2 for x86-on-ARM) often reaches a similar ballpark (often 70-80% of native speed or better on compute-bound tasks). These precedents give confidence that our approach (L1 translator, see §7) can deliver usable performance for legacy binaries, especially since x147's ISA and memory model are designed *from the ground up* to make x86 translation efficient (e.g. TSO memory, similar atomic semantics, etc.).

Photonic Compute Tier (oMAC Islands and Controls)

This is the "photonic heart" of the system: one or more **photonic compute tiles** integrated on the package (via 2.5D interposer) alongside the x147 SoC. Each tile is an optical matrix compute engine specialized for massively parallel linear algebra operations (dense matrix-vector multiplications, convolutions, etc.), which are the core of many AI, graphics, and scientific workloads.

Photonic Tile Architecture:

- Each tile implements a **64×64 Mach-Zehnder interferometer (MZI) mesh** based on a *Clements* design (a specific interferometer arrangement that allows arbitrary unitary transformations or matrix operations)ntt-research.comnature.com. This mesh of waveguides and tunable beam splitters can perform a 64×64 matrix-vector multiply (MVM) in a single pass of light through the chip.
- The photonic tile includes **optical modulators** (e.g. electro-optic modulators) to encode input electrical signals as light intensities or phases, and **photodetector (PD) arrays** to convert the optical output signals back to electrical form. Thus, each photonic tile has an interface: it receives input data (e.g. vector components) from the electronic domain, modulates them onto light, the light propagates through the MZI mesh performing analog multiply-accumulate operations, and the resulting optical signals are detected and converted back to numbers.
- The design is **highly clocked and looped**: the internal photonic computation can run at an effective **≥1 GHz equivalent rate**, with a **deterministic loop latency under 10 ns** from launch to result. In other words, once data is loaded into the modulators, the optical interference and detection completes within about 10 ns. This is orders-of-magnitude lower latency than conventional electronic matrix multipliers of similar size, which take many clock cycles to sequence through the operations. (Indeed, recent research demonstrated an integrated 64×64 optical matrix accelerator achieving ~1 GHz operation with ~3 ns per MAC cycle latencynature.comnature.com. Photonic circuits' latency is fundamentally limited only by the time of flight of light through the chip, which scales linearly with physical size; even for a 64×64 mesh, this can be on the order of a few millimeters of optical path, i.e. a few tens of picoseconds per interferometer stage. Thus, sub-10 ns compute loops are feasiblenature.comnature.com.)
- Each tile has an on-chip **weight SRAM** (static memory) that stores the configuration settings (phase shifts, attenuation coefficients) for the MZI mesh to implement a given

matrix. By writing new values to this SRAM and updating the optical phase shifters (e.g. thermo-optic heaters or electro-optic elements) accordingly, the tile can be reprogrammed to compute a different 64×64 matrix operation. This programming is not instantaneous (it may take microseconds to milliseconds to load new weights and stabilize), so in practice each tile might hold a set of pre-calibrated matrices and switch among them or be reprogrammed during less performance-critical periods.

- **Heater and laser controls:** The tile uses either micro-heaters or electro-optic tuning elements to control the interferometers. It also includes monitor photodiodes ("tap PDs") at strategic points in the mesh to measure optical power levels for calibration and logging. For example, measuring the light leakage or output at certain points can indicate if the mesh is drifting out of alignment. We define a parameter **H (Heaviness)** as the ratio of optical power in vs out, related to signal attenuation; tap PDs allow continuous monitoring of optical loss or drift. If the **H ratio** falls below a threshold (e.g. if ambient optical power in a stage is <10^-3 of expected), the system can flag a calibration alert.
- The tile integrates a dedicated **OPTX DMA engine** and control logic with the SoC. This means the CPU can queue up a photonic operation by filling out a descriptor (pointing to source data in memory, destination for results, which weight matrix to use, etc.) and pushing it to the photonic command queue. The tile's DMA will then fetch the input vector from memory into modulators, trigger the optical computation, and collect the results into memory – all with minimal CPU intervention. The photonic loop being only ~10 ns means even with overheads, end-to-end an optical MVM can be on the order of tens of nanoseconds plus I/O time, which is extremely fast for a 64-wide vector multiply.

"Porch-model" control and calibration: Because photonic computing is analog and subject to physical perturbations (temperature changes, wavelength drift, etc.), robust control schemes are built in hardware:

- **Aperture presets:** The optical mesh can be conceptually tuned by parameters like phase offsets (φ, ψ) and coupling ratios (κ_x, κ_y) for the interferometers. We provide hardware presets for certain "aperture geometries" – essentially mode shapes or connectivity patterns – that can be selected quickly. This allows the programmer (or compiler/runtime) to choose a configuration optimized for certain types of matrices (e.g. a sparse pattern vs dense, or certain symmetry) without individually tuning hundreds of phase shifters each time.
- **Heaviness monitoring (H):** The on-tile photodiodes continuously monitor optical signal levels. If they detect the optical throughput dropping (sign of increased loss or mis-tuned interferometers), the hardware can raise an interrupt or automatically initiate a recalibration sequence. Ensuring the **signal-to-noise ratio** stays above a minimum (H_min threshold) is critical because photonic computation is analog and can degrade with driftnature.comnature.com. By keeping "ambient" stray light or loss below 0.1% (10^-3) of the main signal, we maintain computational accuracy.
- **Fresnel "glint" and resonance management:** Optical systems can suffer from unwanted reflections (Fresnel reflections at interfaces) and resonances (especially if ring resonators are used). In this design, we enforce **resonance hygiene**: if ring resonators are used, they are

confined to ancillary functions like filtering specific wavelengths, not in the main compute path where their thermal sensitivity could cause oscillations. The interferometer mesh uses primarily MZIs which are more linear. A small photonic sensor can detect anomalous coherent reflections (“glints”) and inform the calibration system to adjust or damp those modes.

- **WDM and time-bin multiplexing:** Each photonic tile can potentially operate with multiple parallel optical channels. We allow up to **8 wavelengths (λ) in WDM** or alternatively **8 time-bin slots** as parallel lanes per tile. In practice, this means one tile could perform 8 independent 64×64 operations in parallel (if sources and detectors for 8 wavelengths are available), or time-multiplex 8 sequential operations in a pipelined fashion. Calibrating multiple wavelengths adds complexity (each wavelength may need slightly different phase tuning), but greatly boosts throughput. The hardware provides per-lane calibration and the ability to treat each wavelength/time-bin as a separate “lane” in the OPTX descriptor (see §14). This draws on known techniques in photonic computing where using multiple wavelengths can multiply effective throughputnature.comnature.com.
- **Continuous calibration (“ChiL” style):** Because analog photonic circuits inevitably drift (e.g. due to temperature), continuous in-operation calibration is mandatory. We implement a **chip-in-the-loop (ChiL) optimization** approachresearchgate.netresearchgate.net: the system injects small dither signals and uses the tap photodiodes to sense output changes, then algorithmically adjusts phase shifters to correct errors. This runs as a closed-loop in the background, likely at a modest duty cycle (say 1–3% of time, or in between batches of computations). Research has shown that such approaches can maintain very high precision (9-bit+ accuracy) even in large-scale photonic circuits, by rapidly iterating a few times to counteract drift and fabrication variationresearchgate.netresearchgate.net. The calibration data for each tile (ideal settings, temperature sensitivities, etc.) is stored in **calibration capsules** on the ejectable ROM, meaning the factory characterization of each photonic tile can be loaded to guide the calibration algorithm. In essence, the system is always tweaking itself: a necessity since purely “offline” calibration (one-and-done at boot) is not sufficient for analog photonics – the environment will change, and so must the settings.

Overall, the photonic compute tiles serve as ultra-fast, high-throughput linear algebra engines. They operate in tandem with the electronic cores: the x147 CPUs set up tasks and handle control flow, while large matrix math or vector math can be offloaded to the photonic units. This division of labor leverages the strength of each domain: **electronics for complex logic and precise state, photonics for parallel analog compute**nature.comnature.com.

By keeping the photonic operations on-chip (or in-package) and tightly coupling them (via dedicated DMA and coherence), we avoid the major bottlenecks that plague external accelerators: data movement. Data is fed to the photonic tile over a short, wide interface at extremely high bandwidth (through HBM or on-interposer links), and results come back quickly, minimizing time spent moving data around. This addresses one of the biggest challenges in accelerators – often the *time to move data* dwarfs the *time to compute*. Our optical fabric (next section) further extends this high-bandwidth philosophy across multiple chips.

Optical Fabric (High-Speed Interconnect)

Beyond the single-socket integration of photonics, the architecture includes an **optical backplane fabric** to interconnect multiple nodes (CPUs, GPU accelerators, memory expansion boxes, etc.) at very high speed. Each x147 SoC provides 1–2 optical fabric ports, each capable of on the order of **1–4 Tb/s** of throughput. The fabric is conceptually similar to a supercharged version of existing coherent interconnects (like InfinityFabric or NVLink or CXL), but implemented with photonic links to achieve far higher bandwidth and low latency over longer distances.

Key features of the optical fabric:

- **Credit-based flow control:** The fabric uses a credit or token system to manage buffer occupancy and ensure reliable delivery without overwhelming receivers. This is common in high-speed interconnects; here it's especially important because of the huge bandwidth – a misbehaving sender could flood a receiver with data if not throttled. Credits ensure that senders only transmit when the receiver (or intermediate switch) has buffer space.
- **Fixed latency fast path for small transfers:** For certain short messages or synchronization signals (for example, a cache line write or an acknowledgment), the fabric can bypass some of the heavy protocol overhead and use a cut-through photonic path that is fixed-latency. Essentially, for small payloads the network can operate more like a circuit, giving deterministic low latency (potentially a few nanoseconds plus propagation). This is valuable for things like closing control loops between chips (e.g., coordinating two photonic tiles in different sockets for a larger matrix multiply, or quick coherence messages). The photonic nature (no RC delay, speed of light transmission) means latency over, say, tens of centimeters could be only a fraction of a nanosecond per cm. (For reference, ~20 cm of fiber is ~1 ns of propagation).
- **Memory-class coherence and semantics:** The optical fabric is designed to carry cache-coherent memory traffic between nodes. It functions akin to CXL 3.0 or similar protocols, where devices can share memory regions with coherence. Each port can thus connect to either another CPU, a memory expansion, or a distributed shared memory system. In effect, you could have a pool of optical memory modules and multiple processors all in one coherent address space over the photonic fabric. CXL 3.0 introduced capabilities for memory pooling and sharing across compute nodesbusinesswire.combusinesswire.com, and our fabric builds on that concept with much more bandwidth. We include adapter logic such that, from the perspective of the CPU, accessing remote memory over the optical fabric looks like a NUMA node or a CXL memory device – albeit one with incredibly high transfer rates.
- **Bandwidth and scalability:** Each port at 1–4 Tb/s means, for example, a dual-port CPU could have up to 8 Tb/s of off-package bandwidth. To put that in perspective, 8 Tb/s is 1000 GB/s – several orders larger than a typical CPU's PCIe bandwidth today. Technologies in development already show this is feasible: for instance, **Ayar Labs** demonstrated a 1 Tb/s optical chiplet link as early as 2021optica.org, and is now unveiling 8 Tb/s optical I/O chiplets using 16 wavelengths and the UCIe chiplet standardbusinesswire.combusinesswire.com. Those real-world examples give confidence

that our assumed 1–4 Tb/s per port is attainable with state-of-the-art photonics. We likely use a coarse WDM approach – e.g., 64 fibers × 64 Gb/s each (multiplexed), or fewer fibers with dense wavelength multiplexing – to achieve these rates.

- The optical backplane can be configured in different topologies (point-to-point, switched fabric, or optical ring network between multiple sockets). Early implementations might be direct point links (e.g., a 2-socket system connected by two optical links). Over time, one can imagine optical switches enabling many nodes (similar to how InfiniBand or Ethernet switches work, but optical). Co-packaged optics and optical circuit switching tech are actively being developed to allow dynamic reconfiguration of optical networks in datacenterscambridgeconsultants.comcambridgeconsultants.com.

The main **benefit** of the optical fabric is to remove bottlenecks in scale-up and scale-out scenarios. If one socket's photonic tile isn't enough, two sockets can work together, sharing data over the fabric fast enough that it almost feels like one chip. For example, distributing parts of a large matrix multiplication across two photonic tiles on two sockets is viable if the inter-socket link can supply data as fast as each tile consumes it. Traditional electrical interconnects might choke on that, but an optical 2–4 Tb/s link might handle it. This is particularly relevant for accelerating things like AI models (which are often memory and bandwidth-bound). By prioritizing the **interconnect early in the development**, we acknowledge that many performance wins in modern systems come from moving data faster between compute units, not just speeding up the compute in isolation. In other words, **removing I/O and inter-node bandwidth bottlenecks** can yield more system-level speedup than just cranking FLOPs, especially for distributed workloads.

Memory System and the Free-Form Memory Panel

Physical Memory Architecture: The system uses a hierarchy of memory technologies to balance bandwidth and capacity:

- On-package **HBM (High Bandwidth Memory)** stacks providing extremely high throughput (each stack can provide on the order of 1 TB/s of bandwidth; having 4–8 stacks could yield 2–6+ TB/s aggregate). HBM is used for bandwidth-critical workloads, feeding both the CPU, GPU cluster, and photonic units. It might serve as a unified near memory.
- Traditional **DDR5/DDR6** DRAM channels attached off-package, offering larger capacity (hundreds of GB, up to 1 TB or more total) but with lower bandwidth per channel (perhaps 50–100 GB/s per channel). DDR provides the bulk of capacity for less bandwidth-intensive data.
- **VRAM** (for the GCU) is either a dedicated GDDR memory connected via an optical interposer link, or possibly carved out of HBM if integration allows. (It's conceivable to have a portion of HBM dedicated to GPU use for simplicity.)
- There is also some on-chip **SRAM scratchpad** used by the photonic OPTX engine for staging inputs/outputs or buffering streaming data in/out of the optical mesh. This SRAM might be used for batching multiple small optical operations into one go, etc.

The Bandwidth Arbiter: A critical piece in the memory subsystem is a programmable **Bandwidth Arbiter**, which governs how different clients (CPU cores, the GPU cluster, the photonic tiles, and the optical fabric I/O) share the memory bandwidth (especially HBM and DDR). Instead of leaving this to fixed hardware policies, we expose controls to the system software (and ultimately to users/admins via the panel). The arbiter can enforce:

- **Weights and caps per client:** For instance, one can assign weights indicating proportional share of memory bandwidth each client should get when contending. One might give the photonic accelerator a high weight if running an ML training workload, or give the GPU a high share for a graphics-first workload. Hard caps can also be set to limit a client to at most X GB/s if needed.
- **Queue shaping:** Different traffic classes can be handled differently. For real-time photonic loops (like a quantum error correction decoder that must run in real-time), we can enable cut-through or priority handling of memory accesses – ensuring that, say, the next data needed for an optical computation arriving every 10 ns is always fetched on time. Other traffic can be buffered or rate-limited in token buckets to smooth out bursts and avoid interfering with the real-time streams.
- **Traffic isolation and QoS:** The arbiter ensures that a noisy CPU stream (like a background OS job or a rogue DMA) cannot starve the photonic tile of HBM bandwidth, and vice versa. This is analogous to features in modern CPUs like Intel's Resource Director Technology which provide control over shared resources like memory bandwidthintel.comintel.com. Indeed, Intel's Memory Bandwidth Allocation (MBA) is a precedent: it allows setting an approximate limit or portion of memory bandwidth per workload to manage interferenceintel.comintel.com. Our system generalizes this across multiple memory types and adds the optical fabric as another resource to allocate.

The Free-Form Memory Panel: We expose these arbiter controls through a user-facing interface – both a **physical front-panel** (perhaps a small touch display or dial array on a server chassis) and a **command-line (CLI) tool** for scriptability. The idea is to give operators and developers real-time insight and control over memory and fabric bandwidth distribution, rather than burying it in driver heuristics. The panel allows:

- **Pre-set Profiles:** e.g., *Compute-first*, *Graphics-first*, *AI Training*, *Real-time Photonic (QEC)*, *Balanced*. Each profile corresponds to a particular configuration of weights and priorities. For example, "Graphics-first" might give the GCU a larger fraction of HBM and allocate more optical fabric credit to GPU-related traffic (if doing multi-GPU).
- **Manual sliders/knobs:** The user can manually set, say, "HBM: CPU 30%, GCU 30%, OPTX 30%, Fabric 10%" and the arbiter will enforce roughly those shares (when contention exists, it schedules such that over time those percentages hold). DDR might likewise be partitioned, though DDR is likely less contended if HBM offloads the heavy bandwidth tasks. The optical fabric credits can be set – e.g. guarantee the photonic accelerator 200 Gb/s of the fabric for its traffic, reserving it so that if other traffic tries to flood, they cannot take that reserved portion.

- **Telemetry:** The panel displays live metrics like current HBM bandwidth used by each client (in GB/s), DDR bandwidth usage, optical fabric utilization (how many Gb/s each is sending/receiving), queue depths, and even photonic tile health metrics (e.g., H ratio or calibration activity). Perhaps it shows if any client is hitting a cap or if any optical lane is approaching saturation. Rolling averages (1 s, 1 minute) could be shown for trend analysis. This transparency helps users understand how the photonic accelerator is performing relative to the CPU, if something is bottlenecked on memory, etc.
- **Transactional updates:** Changes made via the panel (or CLI) are not applied abruptly in the middle of operations in a way that could cause instability. They are staged (shadow registers) and then committed at safe points (like a vsync or frame boundary, or at an epoch boundary for computations) to avoid sudden glitches. This ensures consistency and avoids, for instance, oscillations or packet loss if fabric credits were changed mid-stream.

Essentially, the **Free-Form Memory Panel** is about turning typically static hardware resource allocation into something tunable and observable. This is increasingly important in heterogeneous systems – similar in spirit to how datacenter operators use QoS controls to meet SLAs. By making it a first-class feature, we hope to make the system robust under mixed workloads (you can prevent a bandwidth-hungry job from interfering with a latency-critical one by tweaking these controls). Many of these concepts follow known practices of resource partitioning in high-end CPUsintel.comintel.com, but here it's extended to new resources (HBM, optical links) and made accessible.

Ejectable ROM Modules (Secure Boot and Photonic Calibration)

Modern systems often have a boot ROM or flash that holds the initial code for secure boot. Here, we introduce a twist: an **ejectable ROM bay**, i.e., a physically removable module (like a rugged cartridge) that can contain firmware and calibration data. This design serves several purposes:

- **Root of Trust & Recovery:** The system has a built-in **Boot ROM** on the board (immutable code in mask ROM or eFuses) that implements the very first stage of boot and root-of-trust verificationmicrochip.commicrochip.com. This on-board ROM will verify digital signatures on any further code before executing it (establishing secure boot). After that, the system can hand off control to firmware on the **removable ROM** if it is present and valid. The removable ROM can carry a known-good boot image or recovery environment. In case of corruption of the main firmware or if an update goes bad, an admin can insert a recovery cartridge to boot the system in a safe mode.
- **Photonic calibration "capsules":** Each photonic compute tile has certain calibration data (phase offsets, preferred bias settings, trim values for lasers/modulators, etc.). These are determined at manufacturing and might be updated over time. The removable ROM can store these calibration matrices and even field recalibration packages (hence calling them capsules). Because the photonic components may age or drift over years, updated calibration

data might be issued by the vendor – rather than reflashing the system, an operator could plug in a new calibration ROM or update the one in the bay.

- **Field Service and Security Keys:** In sensitive environments, one might not want certain critical data (like the root keys, or certain algorithms) stored permanently in the system. An ejectable module could be required to be present to enable certain functionality (like a physical key). For example, military or industrial users might remove the module when the machine is not in use, ensuring the machine cannot boot or operate without it. The module can be keyed so only authorized ones fit (preventing someone from inserting a random device). All code or data on it is cryptographically signed and measured before usemicrochip.commicrochip.com – unauthorized or tampered cartridges will be rejected by the root-of-trust.
- **Multiple slots:** The design could allow multiple ejectable ROMs (say ROM-0, ROM-1, ...). ROM-0 might be a vendor-supplied module with the canonical secure boot block (only updateable by the vendor, containing, e.g., the x43 hypervisor core and minimal drivers). ROM-1..N could be user slots where you can load alternate firmwares or tools (for instance, a diagnostic monitor, or an alternative OS kernel, or specialized photonic test routines). All these images would of course need to be signed (by either the vendor or possibly by the owner if they manage their own keys) and will be verified against allow-lists on insertion. We also include a **revocation mechanism**: e.g., if a certain ROM image version is found vulnerable, its ID can be added to a revocation list (which could be distributed in a firmware update and stored so that the Boot ROM knows not to boot it)microchip.commicrochip.com. This is in line with NIST Platform Firmware Resiliency guidelines for being able to revoke bad firmware and maintain recovery imagesmicrochip.commicrochip.com.
- **Form factor:** The ROM cartridges are envisioned not as flimsy SD cards but as robust industrial modules (perhaps similar to the old PC card or military grade removable media). They use a SPI or eSPI interface when inserted (the Boot ROM communicates with them over SPI). The connector is physically keyed to ensure only compatible modules seat correctly (and perhaps to satisfy tamper resistance). They could also include a physical latch or lock mechanism.

Security model: Only one ROM module is accessed at a time (or at least, Boot chooses one path). Hot-swapping is allowed only when the system is in a quiescent state (you wouldn’t yank it during normal operation except in a maintenance mode). The ROM controller hardware will refuse to map in a new module’s contents on the fly without a proper reset or secure handoff, to prevent any chance of glitching the system by swapping modules at runtime.

All code coming from the ejectable ROM is **measured and verified** by the immutable boot code before executionmicrochip.commicrochip.com. The presence of these modules does not weaken security – if anything, it strengthens flexibility while maintaining trust, because you can always fall back to a known-good state (by inserting the factory ROM, for example). The use of cryptographic signatures (ECC or RSA) and possibly physical unclonable functions (PUFs) on the board for key storage means the system will only execute authentic firmwaremicrochip.commicrochip.com.

Use cases: Aside from secure boot and calibration, these ROMs could carry things like:

- Alternate **x43 kernel** builds (maybe a real-time optimized build vs a debug build).
- A **diagnostic shell** that one can boot into for low-level testing of photonic hardware (especially useful if the main OS is not coming up).
- **“Safe mode”** firmware that disables photonics and runs the system in a minimal state (for troubleshooting if, say, an optical component failure was crashing the system).
- **Licenses or cryptographic keys** for certain features (one could ship feature unlocks as a physical key – a bit like how some high-end software or hardware require a dongle).
- It's also possible to include **Quantum/PRNG seeds** or other entropy sources in a module, if needed, separate from the main system.

In summary, the ejectable ROM concept adds a **layer of resilience and serviceability**. It’s reminiscent of some older systems that had service processor cartridges or removable BIOS chips, but updated for modern secure boot practices. By making it cryptographically secure, we ensure it doesn’t become an attack vector. By making it removable, we ensure the system can always be recovered or upgraded even if on-board flash is corrupted or if the system is in the field without network access (just ship a new ROM and plug it in to update, etc.).

x43 Kernel and Backward Compatibility Strategy

At the core of the system software is the **x43 kernel**, a custom operating system kernel tailored for this heterogeneous platform. The philosophy of x43 is a minimalistic **micro-hypervisor**: it provides just the essential primitives to securely manage hardware and isolation, while higher-level OS services (drivers, filesystems, etc.) run in user-space for flexibility. Additionally, x43 is designed to handle the unique real-time needs of photonic computing.

Kernel design highlights:

- **Micro-hypervisor architecture:** The kernel runs in S-mode (supervisor) on x147 and primarily does scheduling, memory management, and hypervisor duties. Device drivers, filesystem, networking stack, etc., are in user-mode or in privileged but isolated domains (similar to how seL4 or QNX or other microkernels operate). By keeping the kernel small, we reduce the chance of bugs and make certification/testing easier, which is important given the novel hardware.
- **Real-time lanes:** We partition execution into at least two scheduling classes:
    - A **Real-Time (RT) lane** dedicated to time-critical photonic tasks. This lane is used for things like quantum error correction decoders, or any loop that must run with tight deadlines (tens of nanoseconds scale). Tasks in this lane are given highest priority and are guaranteed not to be paged out or interrupted by non-RT tasks. The kernel ensures these tasks have pre-allocated resources (pinned memory, locked cache lines perhaps, and a reserved slice of compute) so that they can meet their deadlines. Essentially, this is analogous to having a real-time micro-scheduler for

    certain threads that need to wake up, feed the photonic hardware at precise intervals, and process the results (like reading out syndrome measurements, etc.).

    - A **Best-effort lane** for everything else (normal applications, background processes, etc.). This could still be multitasking, time-shared, etc., and even include an entire Linux userland or a Windows compatibility layer, but it will never preempt the RT tasks in a way that violates their deadlines.

- The kernel likely has a notion of **budgeting photonic operations** – e.g., it will coordinate so that two user processes don't try to use the photonic tile at the same time in conflicting ways. It could enforce that certain photonic ops are admitted (like a GPU driver does with GPU jobs) to ensure quality of service.

Now, on to the crucial aspect of **backward compatibility with x86/x64 software**. We acknowledge we cannot run x86 code directly on the x147 cores (different ISA), but we still want to support legacy applications. We provide three tiers of compatibility, which we will also clearly communicate to users/developers:

L0: Source-Level Porting (Native) – This is the *ideal* and highest-performance path. If users have the source code, they can recompile or port their software to run natively on x147 (using provided toolchains like LLVM/GCC for x147 and standard libraries). This yields the best performance and full access to new features (like OPTX instructions). Most high-performance computing, AI, and graphics software can likely be recompiled with minimal changes, as x147 will have a robust POSIX-compatible environment for user programs. We encourage this route for anyone who can manage it, as it avoids any translation overhead. (Guidance: if you need the absolute **predictable performance** or real-time guarantees, **rebuild for x147**).

L1: User-Mode Binary Translation (Dynamic JIT) – This is the primary compatibility mode for most legacy apps where source isn't available or to ease migration. It works as follows:

- An **x86-64 user-space emulator/JIT** runs in user mode on x43. This emulator translates x86 instructions to x147 instructions on the fly. It can be thought of akin to QEMU user-mode or Rosetta 2, but with assistance from hardware and the OS for performance.

- System calls made by the x86 program are trapped and translated to the x43 kernel ABI (similar to how WoW (Windows-on-Windows) or Linux compatibility layers work). For Linux apps, we can implement a syscall translator that maps Linux x86 syscalls to x43 kernel calls. For Windows applications, we plan to utilize a *Wine-like* approach: provide an implementation of the Windows API (NTDLL and Win32 libraries) that runs on x43, so that many Windows apps can run as if they were on Windows, even though the kernel underneath is x43.

- The translator uses various optimizations: it caches translated blocks of code (our μop cache in hardware can store these translations for reuse, as mentioned). It also leverages the fact that x147's vector registers can directly handle SSE/AVX instructions – the translator will map x86 SSE/AVX ops to x147 VX ops one-to-one where possible, meaning vectorized x86 code runs very efficiently.

- Memory model alignment (TSO) means x86 memory ordering is naturally respected, so we don't need to add extra fences in most cases – this preserves performance and correctness.
- This mode covers standard 64-bit apps and even 32-bit apps (x86-32) if needed by translating their code as well (32-bit will be a bit trickier due to things like segmentation, but it's doable in the translator with some overhead).
- For graphics and GPU-heavy applications (like games), we integrate translation layers such as **DXVK** or **VKD3D** for translating DirectX calls to Vulkan, which our native graphics stack can handle. Shaders or GPU compute kernels in those apps can also be intercepted – e.g., if a game uses D3D or OpenGL shaders that involve heavy matrix multiplications, those might be auto-detected and offloaded to OPTX for acceleration. This way, even some legacy games or GPU programs could see a boost by transparently using the photonic hardware for specific operations.
- Performance expectations for L1: We anticipate somewhere around **0.6–1.0× native** x147 performance on compute-heavy code (i.e., some overhead, but thanks to caching and optimization, sometimes close to native if the code spends most time in tight loops that we can optimize)blog.kowalczyk.infoblog.kowalczyk.info. For system-call heavy or very pointer-intensive code (lots of context switches, or weird self-modifying tricks), performance might drop to 0.3–0.5× native – these are worst-case. But many workloads (think SPEC CPU or common applications) should run quite decently under L1.

L2: Full System Virtualization (VM) – For cases that absolutely require a full x86 environment (including kernel-mode x86 code, drivers, or obscure OS behaviors), we provide the option of running a full virtual machine. Under H-mode on x147, our hypervisor can instantiate a VM that thinks it's running on an x86 machine. This VM would use dynamic binary translation in the hypervisor to execute the guest OS code (similar to how VMware or VirtualBox can emulate x86 on ARM with JIT, but here possibly with some hardware assists).

- This is the slowest but most compatible approach. It's essentially like running QEMU with JIT for the whole OS. We'd use it for things like legacy drivers that have to run in Windows kernel mode, or an OS that we can't modify at all.
- We will however paravirtualize where possible: e.g., provide virtio devices or enlightened interfaces to reduce overhead of I/O.
- We might also allow the VM to use the L1 translator internally for usermode, to speed up user-space within the VM (i.e., the VM's kernel runs under translation, and it in turn runs its user apps via a similar mechanism as L1). This stacking is complex but doable.
- Performance could be on the order of 0.2–0.5× of native at best for heavy workloads due to double translation overhead and VM overhead. It's really a last resort for compatibility, not for performance.

By clearly delineating these three levels, we can **be honest with users** about what will run and how well:

- If an application can be recompiled, that is always best (L0).

15

- If it’s a typical app or game not doing crazy kernel stuff, L1 will run it with acceptable performance.
- If it’s something like a special driver or needs real Windows kernel, use L2 but expect a bigger slowdown.

Notably, we explicitly **do not support** certain things natively:

- **No x86 ring-0 native execution:** i.e., you cannot take a Windows kernel or an old Linux kernel and run it on the bare metal – it has to be in a VM. This prevents a whole class of issues and also simplifies our hypervisor (we don’t have to accommodate x86 privileged behaviors in hardware).
- **No Intel SGX or TSX emulation:** those are special Intel features (enclaves, transactional memory) which we won’t support. If an app uses them, on our system it will either fail or run in a VM with those features effectively absent.
- **16-bit real mode or unusual CPU modes:** Those will only be supported inside VMs (for example, DOS programs could be run in a VM that virtualizes an old PC). The new ISA has no real-mode equivalent, it’s purely 64-bit.

We will publish **compatibility guidelines** summarizing this:

- For predictable performance or real-time, use native x147 (recompile).
- For most Linux software, the user-mode translation (plus a Linux-compatible subsystem in x43) will handle it fine.
- For Windows applications, we’ll provide a Wine-like environment under L1. For those that don’t work with Wine, an actual Windows VM under L2 is the fallback.
- If someone needs to run a legacy kernel driver (say for a specific device), they’ll have to do it in a VM – perhaps with device passthrough if needed.

This strategy is inspired by prior art: Apple’s macOS on Apple Silicon uses a similar tiered approach (Rosetta 2 for apps, virtualization for Windows if needed), and historically Transmeta and others have proven out the dynamic translation concept. By implementing it at the OS/hypervisor level with cooperation from hardware (µop cache, TSO memory, etc.), we expect a fairly smooth experience for end-users, who might not even notice their app is running under emulation in many cases.

OS Services and Developer Model

Building atop the x43 micro-hypervisor, the developer environment will look familiar in many ways (especially if using L0 or L1):

- We aim to support standard **POSIX** APIs for native apps. Developers can use familiar compilers (we’ll upstream x147 support to LLVM/GCC) and link against standard C/C++ libraries (glibc or musl) ported to x147. So developing a Linux-style userland on this system should be straightforward.

- For those targeting photonic acceleration, we provide a library and runtime called **libOPTX**. This library offers a high-level API to applications to offload computations to photonic tiles. Instead of directly writing assembly for OPTX, a developer can call functions like `optx_enqueue_mvm(matrix, input_vector, output_vector, opts)` which behind the scenes prepare the OPTX descriptor (see §14 for an example of such a descriptor structure) and submit it to the hardware. The library could handle things like splitting larger matrices into 64×64 tile operations, coordinating multiple photonic tiles if available, and managing synchronization with the CPU.
- For machine learning frameworks, we will provide integration at the graph compiler level. For example, an **XLA** backend (for TensorFlow/JAX), an **ONNX Runtime** execution provider, and a **PyTorch** device (maybe `torch.device('optx')`) will be developed. These will recognize operations like big matrix multiplies, convolutions, transformers' attention mechanisms, etc., and route them to the photonic backend. If an operation is not supported or too small to benefit, they can fall back to either the vector units or the GPU (whichever is appropriate). This is akin to how today one might have a GPU backend or an FPGA backend for a framework – here it's a photonic backend.
- Special domain: **QEC (Quantum Error Correction) SDK** – Given the interest in using this for quantum decoder workloads, we'll likely create a small SDK to help coding theorists implement their decoders. This could include templates for common algorithms like belief propagation or neural-network-based decoders that are optimized to run in the real-time photonic lane. It might expose a "syndrome stream" abstraction: you feed in a stream of qubit error syndromes and the photonic tile, in real-time, produces correction suggestions. Having a library for that would help quantum labs to leverage the hardware without starting from scratch.
- **Optical fabric API:** Developers or system software will also have access to APIs to manage the optical network. For example, if an application knows it will frequently use a remote memory node or another processor via the fabric, it could request a reserved channel or certain bandwidth (subject to policy). This might look like a system call or library call to "allocate X Gbps guaranteed on the fabric to Node Y for the next N milliseconds" or to register a memory region for remote access with certain latency requirements. Under the hood, this interfaces with the fabric controllers and the free-form panel settings (if dynamic).
- The overall developer model is heterogeneous: you have CPU code, GPU code (shader or kernel), and photonic code. The photonic code might not be code per se (since the photonic tile is more like setting up data and letting it propagate), but one needs to schedule and orchestrate it. We'll likely provide a task-based programming model where photonic tasks are first-class citizens. Possibly something like a dataflow programming model where you can specify that this matrix multiply goes on photonic tile and then its output triggers some CPU post-processing, etc., with dependency management.

Operating systems support: We expect most users will run a Linux-based environment on this (since x43 can host a Linux-like userland easily, and we could even run a full Linux kernel in a VM if needed). For Windows applications, as mentioned, we rely on a compatibility layer (Wine or a thin NT-compatible environment). We won't run the Windows kernel natively, but perhaps an instance of Windows could run in a VM if absolutely required (with performance cost).

The presence of a hypervisor means we could also allow other OS instances in VMs (for example, a user could launch a vanilla Linux/x86 VM if they wanted a full distro environment isolated from the main system). The hardware virtualization support (H-mode) and dynamic translation would handle that.

Driver development: Most drivers for physical devices (other than our core photonic components) will be in user-space processes, communicating with the hardware via IPC or a small kernel proxy. This follows modern trends (e.g., user-space drivers in seL4, DPDK for networking, etc.). It increases stability (a buggy driver can't crash the kernel, only its own process) and ease of development (you can use standard debugging tools on a user process driver).

Security and isolation: By leveraging a microkernel approach, we can isolate different subsystems. For instance, the photonic control loop (calibration and operation scheduling) can be in a privileged service separate from the main OS, ensuring that even if a userland app misbehaves, it cannot stop the calibration or timing-critical functions. Also, running legacy x86 code in user mode with syscall translation means it's easier to sandbox – an x86 process can be constrained just like any other, and we can monitor its syscalls or limit its capabilities (like seccomp, etc., but now at the translation layer).

In summary, the OS and developer model tries to **marry familiarity with new capabilities**:

- If you just want to compile C++ and run, you can do that (just like on any Linux, perhaps).
- If you want to take advantage of photonics, you have libraries and frameworks to help.
- If you run unmodified binaries, the system will accommodate them via translation.
- You have introspection via the panel and possibly tools to profile how photonic tasks are running, etc.
- We hide the complexity of the photonic hardware behind sane abstractions so developers don't need a PhD in optics to use it, but we also allow power users to tweak things (like aperture profiles or WDM usage flags if they really want to optimize an operation for the photonic tile, they could do so through libOPTX advanced options).

Packaging, Power, and Thermal Considerations

Bringing this advanced system to reality involves challenges in packaging and managing power/thermals:

- **2.5D Integration:** The x147 CPU die and the photonic compute die(s) are integrated on a silicon interposer (much like how HBM is integrated today with GPUs). The interposer provides high-density routing between the CMOS and photonics. This is essential because the bandwidth between the CPU and photonic tile must be huge (to feed data for those 10 ns loops). By having them side by side on an interposer, we can use wide buses or even silicon waveguides for direct optical coupling. (There is research on photonic interposers enabling hundreds of GB/s chiplet connectionstspasemiconductor.substack.comtspasemiconductor.substack.com – Lightmatter’s recent photonic interposer announcement is an example). If HBM stacks are used, those too sit on the interposer adjacent to the CPU.

- **Co-packaged optics for I/O:** The optical fabric ports likely use integrated photonic transceivers co-packaged as well. Instead of routing high-speed electrical signals to the board edge, we have, say, a photonic chip or module next to the CPU that directly converts to fiber. We might expose an optical connector (like an **MXC** connector – a multi-fiber connector used in some datacenter gear) on the motherboard for the fabric links. Each such connector might carry e.g. 16 fibers for transmit and 16 for receive, each fiber carrying multiple wavelengths.
- **Laser sources:** One consideration – photonic chips need laser light sources. Often, lasers (especially high-power ones) are kept off-chip (due to heat and to allow sharing one laser across multiple modulators). We might have an external laser module (maybe part of the optical backplane infrastructure) that feeds light into the photonic tiles via fiber or waveguides. The power consumption of lasers is not trivial, but those might be accounted for separately (the photonic tile’s 10–40 W might assume modulators and detectors but external laser supply).
- **Power Budget:** Rough estimate per major component:
    - x147 CPU cluster: depending on core count (8 to 32) and frequency, perhaps **80–200 W**. (This is in line with modern high-end CPUs in 5nm which easily hit 150W+ with many cores).
    - Photonic tile(s): maybe **10–40 W** each. Photonics itself can be very efficient for computation (no switching transistors for each multiply), but power is used in modulators, photodetectors, and mostly in the **thermal tuning** (heaters) and DACs controlling them. Also high-speed ADC/DAC for I/O of optical signals can consume power. If lasers are included on-package, that could add more, but let’s say lasers are offloaded.
    - HBM memory: 4–8 stacks might consume **20–50 W** total (HBM can draw a lot at high bandwidth).
    - Other I/O (PCIe, DDR links, etc.): maybe another 20–30W if fully utilized.
    - So a full socket might be in the 150–300 W range, which is high but comparable to a high-end GPU or CPU today. It will require robust cooling (likely liquid or high-performance air).
- **Thermal Management:** Photonic devices are **very sensitive to temperature** (phase shifts change with temperature). So the chip design must partition hot zones and sensitive zones. We will:
    - Isolate the heaters (MZI phase shifters often use micro-heaters). They should be thermally isolated from logic as much as possible. Possibly the photonic die is

    separate so heat from the CPU doesn’t directly cause big gradients on the photonic mesh.
  - Use **athermal design** where possible: there are photonic design techniques and material choices (like silicon trenches, or using materials with opposite thermal coefficients) to reduce how much a resonator or interferometer shifts with temp.
  - Plenty of temperature sensors distributed across the photonic tile and interposer to monitor local temps. The calibration loop frequency can be adjusted based on temperature readings (if things are running hot and fluctuating, calibrate more often).
  - **Active cooling** likely – maybe a cold plate that covers both the CPU and photonic tile, but perhaps with sections (the photonic tile might be kept at a constant reference temperature with a TEC (thermoelectric cooler) if necessary, to stabilize it).
  - The optical transceivers for the fabric also need cooling but they might be fine as long as within spec.
- The package likely will be large (like an SoC + 2 photonic dies + 4 HBM = quite a bit). It might resemble an NVIDIA H100 GPU package which has GPU + HBM around it. Here we have CPU + others, possibly on an even larger interposer.
- **Optical I/O connectors:** Each optical fabric port with up to 1–4 Tb/s might use something like a **Silicon Photonics transceiver** coupling to fiber. Connectors like **MXC** can carry 64 fibers (16 Tx, 16 Rx duplicates maybe) and handle a few Tb/s. We’ll design the front panel of a server board to have a couple of these optical ports. Fiber management becomes part of the system integration (bundles of fiber cables connecting racks or modules). This is a bit exotic for typical computers, but it’s already seen in some specialized HPC setups. The advantage is immense bandwidth at potentially lower energy per bit than copper for long reaches.

To summarize, packaging this system is non-trivial but within the frontier of current tech: co-packaged photonics, 2.5D multi-chip modules, and advanced cooling are all known techniques being pursued in 2025 for high-end AI and networking systems. We are essentially combining and requiring the best of those. The result should be a tightly integrated compute module with unprecedented compute and communication abilities.

Build and Bring-up Plan (90 / 180 / 360 Day Milestones)

Developing such a system is complex, so a phased approach is planned, with key milestones at roughly 90 days, 180 days, and 360 days into the project (assuming an aggressive development schedule):

- **Day 90 (3 months) – Prototyping Key Concepts:**
  - We will use an **FPGA-based prototype** of the x147 core (or at least its translator engine) to start developing and testing the x86 binary translation in absence of real silicon. The FPGA will run a simplified x147 core at low speed, enough to boot a

    Linux kernel in emulation and test that x86 programs can run under the JIT translator on x43 (likely using an existing RISC-V or minimal kernel ported to x147).

  - On the photonics side, we set up a **“4F” optical bench** (free-space optics or fiber) to mimic a small MZI network. This is used to validate our calibration concepts: we intentionally induce drift or perturbations and test that our **H monitoring and ChiL calibration loop** can correct it. Essentially a lab experiment to prove <1% error with continuous calibration.
  - A very early **OPTX queue emulator** runs on a host PC: we simulate the photonic tile’s behavior in software (perhaps at a high level) to let developers start writing the libOPTX and see how the programming model feels. This is done on a commodity system with an emulator intercepting calls (like how you’d develop CUDA code on a CPU before having a GPU).
  - The **Free-Form Memory Panel prototype** is built as a simple CLI tool at first, controlling a fake “Bandwidth Arbiter” model. For example, we might simulate two memory clients producing traffic and allow the panel to throttle one or the other. By 90 days, we want a demonstration where an operator can run a script to change memory bandwidth allocation and see the effect in a bandwidth graph. This proves out the concept and helps refine the UI/knobs.
- **Day 180 (6 months) – First Hardware and System Bring-up:**
  - We expect the **first photonic compute tile** prototype in hand, likely a 64×64 MZI mesh fabricated on a silicon photonics process, bonded to a smaller control ASIC (or possibly using a high-speed DAC/ADC board for I/O). We integrate it on a 2.5D test package or module to evaluate basic functionality. The goal is to achieve a closed-loop optical MVM operation at target speed (~1 GHz) on a small scale. We’ll test the modulators, detectors, measure latency, and see the calibration working in situ.
  - A preliminary version of the **x147 core** (maybe one core) in silicon or FPGA gets to the point of booting the **x43 kernel** (at least a minimal subset). We might use an FPGA for the core and a small RISC-V or something for peripheral. By this time, we want “Hello, world” running on x43 on x147, meaning our new ISA can run code.
  - The **L1 JIT translator** should be running basic user-space programs (perhaps SPECint or some simple benchmarks) and demonstrating performance within expected ranges on an emulator or FPGA (it will be slow on FPGA, but we can measure translation overhead in terms of instructions).
  - Basic **graphics stack** might be up via software emulation (maybe using Mesa and a dummy GPU) and importantly, **Vulkan translation layers (DXVK)** functioning in the environment – so that we can prove we can run a simple game or graphic demo through the translation.
  - **OPTX offload tests:** Using the first photonic tile, we attempt a simple end-to-end: e.g., offload a small matrix multiplication from an application running on x147 (or a

21

    surrogate system) to the photonic tile via the OPTX API, and verify correctness and performance vs CPU. Even if all pieces are not final, this is the integration test of the vertical stack.

  - By 180 days, we likely also have the initial design of the optical fabric transceiver. Perhaps a loopback test: send data from one port through fiber and back to another port, verifying the protocol, error rates, and latency. It might not be full Tb/s yet, but we may demonstrate a partial link (say 100 Gb/s) as a proof of concept.

- **Day 360 (1 year) – Full System Demonstrations:**

  - We have prototype **x147 SoC** (maybe not final silicon at full core count, but a test-chip with at least a couple of cores and a photonic tile integrated). Two such prototypes can be connected via the optical fabric link. We then demonstrate a real application across them.

  - One target demo: a **Transformer attention kernel** (which involves matrix multiply for queries/keys/values and a softmax) running distributed across two sockets over the optical fabric. Because attention is communication-heavy (all-to-all between queries and keys), using the optical fabric should speed it up significantly. We measure the speedup vs if the two sockets communicated over a PCIe or slower link.

  - Another target demo: a **Mixture-of-Experts (MoE) routing** scenario, where different expert models reside on different photonic tiles/nodes and high-bandwidth switching is needed to route data to the correct expert. This leverages both the photonic compute and the fabric to show how we can scale AI model throughput.

  - A **Quantum Error Correction (QEC) decoder** demo: for example, take a surface code with, say, a 1000-qubit logical area, generate a stream of error syndromes as if a quantum chip is producing them in real-time. Feed this into our photonic tile running a belief-propagation or neural net decoder. Show that we can output corrections within microseconds, meeting the demands for active error correction. This would be a landmark result because current solutions struggle with that timing.

  - By one year, we also aim to publish the **compatibility guide** and the developer SDK. That likely includes documentation, toolchains, and maybe open-sourcing parts of the system (especially the translator, so community can help optimize it or trust it).

  - Essentially at 12 months, we want to prove that our concept is not only functional but excels in at least one real use-case (probably AI training or inference), and that legacy compatibility is handled well enough to run common software in our environment.

These milestones are ambitious, but they guide the team on what to prioritize:
Focus early on calibration (since photonic drift is a risk), on the translator (since compatibility is a promise we make), and on the fabric (since without fast I/O the whole benefit of photonics could be limited by data movement). Photonics-specific features like maybe adding more optical lanes or fancy analog tricks can be iterated later – first we need to nail the fundamentals that tie the system together.

Risks and Mitigations

Any cutting-edge approach comes with significant risks. We enumerate the major ones and how we plan to mitigate them:

- **Analog drift and reliability:** The photonic computation is analog and will drift with temperature, time, etc. **Mitigation:** as stated, continuous calibration is mandatory. We allocate a small percentage of cycles to constantly measure and adjust. We also incorporate designing the mesh to be as tolerant as possible (e.g., operating points that are less sensitive to drift, redundancy in calibrations, etc.). We will no doubt need a lot of testing to ensure the calibration loop can keep up under different conditions (like a data center HVAC blasting cold air, then stopping – causing a 5°C temp swing; the system should adapt). If needed, we'll slightly over-provision photonic performance so that even if calibration isn't perfect, we have margin to meet spec.
- **x86 performance expectations and communication:** Users might assume "it runs x86, so it's as fast as a normal x86". If some workloads (especially ones heavy in system calls or corner-case instructions) are slower, there could be disappointment or negative perception. **Mitigation:** manage expectations through clear communication (publish detailed performance numbers for L1 vs native, as in the compatibility guide). Provide tools to profile where time is spent so users understand if, say, a particular library call is not yet optimized in the translation. Over time, we can optimize the common slow paths. We will emphasize that **for guaranteed performance, rebuilding for x147 is the way** – this parallels how Apple handled Rosetta (encouraging native builds for best results).
- **Driver and OS ecosystem ("driver hell"):** One of the hardest parts of new hardware adoption is device drivers and software ecosystem. If we don't run Windows natively, what about all the Windows-only applications that need their drivers (like some games with anti-cheat drivers, or professional hardware that only has Windows drivers)? **Mitigation:** By using VMs for those cases (L2), we provide a fallback, though with perf cost. Another approach is to engage with the community to port or implement open drivers for popular hardware (especially GPUs won't be needed externally since we have our GCU, but think of things like USB devices, etc.). Our strategy of user-space drivers can simplify porting drivers because many device drivers could be run in compatibility mode themselves possibly. Still, it's a risk that some things just won't work if they rely on x86 kernel hooks (like certain anti-cheat or low-level software). We may have to simply accept some niche use-cases aren't supported.
- **Interconnect complexity and debugging:** The optical fabric and memory coherence between potentially multiple sockets is complex. Bugs there can be very hard to find (race conditions, etc.). **Mitigation:** Prioritize bringing up the fabric early (as seen in milestones) even if it means delaying some photonic compute features. We believe a large part of the performance gain is in removing bandwidth bottlenecks, so getting the fabric right is crucial. Use formal verification on the coherence protocol if possible, lots of simulations, and have robust monitoring (performance counters and logic analyzer hooks in the fabric to debug issues).

23

- **Legal/IP concerns:** Implementing x86 compatibility opens potential intellectual property issues (e.g., we must ensure our binary translation doesn't use any patented techniques of Intel/AMD in illegal ways, and we aren't using any of their microcode or proprietary firmware). Also, using names like DirectX, etc., might have licensing considerations. **Mitigation:** We'll do a clean-room implementation of the translator, referencing only publicly documented behavior of x86 (plenty of literature on that). We won't use any leaked or licensed microcode. We can also try to leverage existing open-source projects (QEMU, DynamoRIO, etc.) as a base – those have already navigated legal issues. We will publish a compliance document clarifying that we are not an "x86 implementation" per se (which might require a license) but an assistive translation system – basically akin to a software emulator which typically is legal. If needed, we might engage with Intel to ensure we're not treading on anything (noting that companies like Transmeta did similar things; Intel sued them once but settled, if memory serves). As for things like supporting Windows APIs, Wine has shown that reimplementing those is fine legally. And for any open-source components we use, we'll comply with licenses.

There are certainly other risks (market acceptance, cost, etc.), but these are the technical big ones. We intend to be upfront about them to our stakeholders and have contingency plans (for example, if photonics yields are poor, maybe we fall back to an FPGA-based accelerator; if translation is too slow for some critical app, maybe encourage that vendor to make a native version, etc.). The key is **modularity** – the system still functions as a general computer even if photonics underperforms or is off; it would just behave like a normal CPU+GPU machine.

What the Free-Form Memory Panel Actually Exposes

To avoid any marketing fluff around this feature, here's a concrete list of controls and readouts the panel (and corresponding CLI `memctl`) will provide:

- **Profile Selection:** A set of predefined profiles, as mentioned: e.g. `RT-Photonic` (real-time photonic, which would crank photonic tile priority, guarantee fabric for it, etc.), `AI-Train` (assuming a mix of CPU and photonic heavy use, perhaps balanced in favor of feeding photonic tiles continuously), `Graphics-First` (give GPU more love), `Balanced` (equal priorities), maybe `Memory-Test` (to isolate memory for diagnostics), etc. These profiles set multiple parameters in one go.
- **HBM Share Allocation:** You can manually set percentages or weights for each major HBM client. For instance: `CPU=40, GCU=20, OPTX=30, Fabric=10` (which would then normalize to 100%). This means if contention arises for HBM bandwidth, the arbiter tries to enforce that division. (If some client isn't using their full share, others can utilize the slack, of course).
- **DDR Share Allocation:** Similarly for off-chip DDR. Since DDR is slower, maybe the photonic tile isn't even using DDR (it would mostly use HBM for speed), so DDR might just be CPU vs general I/O.

- **Optical Fabric Credits:** A setting to reserve a portion of the fabric for certain traffic, specifically the photonic real-time loop. For example, if we have 1 Tb/s link and we know the photonic QEC loop needs 200 Gb/s worst-case, we reserve that. The panel can allow something like `fabric reserve OPTX=200G` as in the CLI example. It could also allow reserving for GPU or others if needed. If not used, reserved portion can potentially remain idle (or we might allow it to be borrowed by others but preemptible).
- **Hard Limits (Caps):** The admin can cap a component to at most X bandwidth to prevent it from ever starving others. For example, cap CPU memory to 100 GB/s if we suspect a rogue CPU thread could otherwise saturate 1 TB/s HBM. The arbiter schedule is work-conserving and starvation-free (likely using weighted fair queuing or deficit round-robin with weights).
- **Latency/QoS settings:** Not directly a slider, but possibly an option to give certain traffic low latency treatment (this ties into the "cut-through for OPTX" idea). The panel might have an on/off for "Real-time mode" which, when on, ensures any photonic DMA or fabric packet is treated at highest priority through all queues (with potential negative impact on latency for others).
- **Telemetry Displays:** Continuously updated metrics:
    - Per client HBM bandwidth in GB/s (maybe both instantaneous and average).
    - DDR bandwidth similarly.
    - Fabric utilization (each direction).
    - Photonic tile queue depth (are we feeding it enough? Is it starving for data or always waiting?).
    - H ratio or any optical warnings (if the photonic tile's monitors indicate loss or misalignment, maybe show a warning or an indicator LED).
    - Calibration status: e.g., "Calibrating: 2% duty, stable" or "Calibration needed!" if something is off.
    - Perhaps memory latency stats (if one component is causing increased memory latency for others, maybe show that).
- **Interface:** As a physical front panel, one could imagine a touchscreen where you can tap a profile or adjust a slider for percentages. For CLI/GUI, `memctl` commands as given in the example allow scripting these changes or integrating with cluster management software. It's important that this not be just a gimmick; we want power users to actually use it (e.g., data center operators could dynamically adjust profiles based on workload schedules, or even an AI could tune it).

In essence, the Free-Form panel turns resource management into a *first-class interactive feature*. This is somewhat novel – most systems have such resource controls buried in MSRs or cgroups configurations. We surface it so that, e.g., a researcher can quickly put the system in "QEC real-time mode" when running quantum experiments, and then back to "AI training mode" when doing offline training at full throughput.

25

And importantly, the panel is **not just blinkenlights** – it's a serious control interface. The mention "not just pretty lights" emphasizes that it's meant to deliver real control, not just a dashboard.

Backward Compatibility Policy (For Publication)

We will publish a clear policy/doc for customers regarding x86/x64 software compatibility on this platform, summarizing much of what was described in §7:

- **Compatibility Tiers:**
    - **L0 (Native Recompile):** Full performance, requires recompiling to x147 ISA. Recommended for all performance-sensitive or new development. (Analogy: like running native code on any new architecture – best case scenario).
    - **L1 (User-space JIT Translation):** High-performance translation of x86-64 *applications* to x147. Supports Linux binaries (ELF64) out of the box. Windows applications supported via a compatibility layer (no need to recompile the app, but they run on our provided Win32 environment). Offers near-native speed on CPU-bound workloads, somewhat lower on others. This is the default mode for most software you have that you can't recompile.
    - **L2 (Full VM Emulation):** Complete x86 system emulation for legacy OS and drivers. Use when you absolutely must run an OS or driver that isn't provided in L1. Performance is substantially lower and overhead is high, so use sparingly (basically only if L1 is not sufficient).
- **Performance Expectations:** (We will likely provide a table or ranges)
    - L0: **1.0x** (baseline, that is native).
    - L1: For computation-heavy apps (e.g., SPECint/fp, games without many syscalls), typically **0.6–1.0×** of native x147 speed. For apps heavy in OS interactions or those using less JIT-friendly instructions, maybe **0.3–0.5×**. The JIT has a warm-up time, so short-running processes might see more overhead initially.
    - L2: Varies widely by workload, but generally **0.2–0.5×** of a native x147 (or even worse in worst-case). Essentially, assume it's slow and use only if needed for compatibility (e.g., some enterprise might use this to run a specific Windows-only software that doesn't work under Wine).
- **Supported Operating Systems/Environments:**
    - Native: The system itself runs the x43 micro-hypervisor and a Linux-based userland (we might brand it differently, but fundamentally it's a Linux environment). So you can run Linux applications (either recompiled or via L1).
    - L1: We explicitly support Linux/x86-64 binaries (probably glibc-based, etc. – basically most mainstream Linux distros' binaries should work). We also support Windows userland via our builtin layer (which we will keep updated to handle commonly used Win32 and DirectX APIs, etc., possibly based on Wine). We do **not**

    support running arbitrary macOS binaries or other OS binaries – those would likely not run without an equivalent layer.

    - L2: In a VM, you can install any x86 OS: Windows, Linux, *BSD, DOS, etc. We'll provide the virtualization infrastructure (likely modified QEMU). However, we might not support every device in those VMs; we'll focus on virtio and basic devices.

- **Explicit Non-Supported Features:**
    - **Native x86 kernel code:** as said, you can't run an x86 OS on the bare metal, only in a VM.
    - **Intel-specific CPU features:** We do not support things like Intel SGX secure enclaves or TSX transactional memory. If an app tries to use them under L1, it will likely get an illegal instruction or be handled in software with no effect. Under L2, it will behave as if the CPU doesn't have those features (just like some older CPUs or AMD CPUs wouldn't).
    - **16-bit and obsolete modes:** Real-mode or V86 mode code will only work inside a VM that virtualizes an old PC. We don't handle that at L1 at all (very few modern apps need it, except maybe some installers or DOS programs).
    - Probably mention that certain performance counters or low-level stuff in x86 won't be present – but that's detail.

Essentially, we want users to know: **it's not an x86 processor, but it will *run* your x86 software in most cases through translation**. We'll likely name the translator something and market it, but also be honest that for best results, port to native.

We might include a recommendation like: if you're a software vendor, please consider adding support for x147 in your build pipeline; it's straightforward (LLVM/GCC support it) and your software will then directly tap into photonic acceleration and full performance.

Concrete Interfaces and Examples

To make things more tangible for developers, we provide examples of the key interfaces.

OPTX Descriptor Example: This C-like struct (64 bytes) defines a single photonic matrix-vector operation:

```
struct optx_mvm_desc {
  uint64_t src_vec_addr;   // Device address of input vector (must be pinned in memory)
  uint64_t dst_vec_addr;   // Device address where output vector will be stored
  uint64_t weight_id;      // Identifier for which weight matrix (64x64) to use (pre-loaded in photonic
tile)
  uint32_t m, n;           // Dimensions of the operation (<= 64 for each, per single launch)
  uint16_t lanes;          // Number of parallel lanes (WDM channels or time slots) to use
  uint16_t aperture_id;    // Selects an aperture preset (tuning profile) for this operation
  uint16_t h_min;          // Minimum acceptable H (signal strength) ratio; triggers recalibration if
violated
```

27

```
  uint16_t flags;          // Flags for activation functions, saturation behavior, etc.
};
```

A user-space program (or a driver library) would fill out such a descriptor, then either do an `ioctl` or a hypercall to queue it to the photonic engine. Multiple descriptors could be chained for batched execution. The hardware, upon processing this, will set up the modulators with the input vector (of length n), ensure the weight matrix is loaded/tuned for the given ID (matrix size m×n, potentially n==m==64 for full size), use the specified aperture settings and maybe only certain wavelengths (lanes), perform the optical MVM, and write the result to the output address.

This design allows flexibility: you could launch smaller operations (e.g., a 16×16 multiply) by specifying m=16,n=16 and it will effectively use a 16×16 sub-section of the mesh or just ignore the rest. Or you can use multiple lanes to do, say, eight 64×64 multiplies in parallel if the data and weight buffers are set accordingly.

Free-Form Panel CLI Example: As given, the `memctl` utility syntax:

```
# Set a preset profile
memctl profile set AI-Train

# Manually adjust HBM shares (here CPU=30%, GCU=30%, OPTX=30%, Fabric DMA=10%)
memctl hbm set CPU=30 GCU=30 OPTX=30 FAB=10

# Reserve 200 Gbps on the optical fabric for photonic (OPTX) traffic
memctl fabric reserve OPTX=200Gbps

# Display live telemetry
memctl show –live
```

The output of `memctl show --live` might look like:

```
Profile: Custom (modified from AI-Train)
HBM Bandwidth (last 1s average): CPU 0.8 TB/s, GCU 0.7 TB/s, OPTX 1.5 TB/s, FabricIO 0.2
TB/s (Total 3.2 TB/s of 4.0 TB/s)
DDR Bandwidth (last 1s): CPU 25 GB/s, GCU 5 GB/s, OPTX 0 GB/s, FabricIO 10 GB/s
Fabric Utilization: TX 180 Gbps (90% of reserved 200 Gbps), RX 50 Gbps
OPTX Tile 0: Queue Depth 2, SNR (H) 0.85 (within limits), Calibrating: 1% duty
OPTX Tile 1: Idle
Temperature: CPU die 75°C, Photonic die 50°C (stable)
```

This shows the kind of info available. An admin could watch this while adjusting and see the effect.

We will also have an equivalent GUI for those who prefer a dashboard. Possibly integration with existing system management (like IPMI or Redfish metrics for datacenters).

ISA and Assembly: We might also publish a bit of example assembly of x147 to show its cleanliness vs x86, but that might be too low-level for most. However, developers of compilers will get the ISA spec documentation.

Where We Draw the Line (Non-Goals and Exclusions)

It's important to state what this architecture is *not* going to do, to avoid wasting time or confusion:

- **No optical RAM:** We are not attempting to create optical random-access memory or storage. All main memory is electronic (HBM, DDR, etc.). Optical computing is used for fast matrix operations and communication, but for storing arbitrary data with random access, optics just doesn't make sense (photonic memory is either non-existent or far inferior in random access capability to electronic memory)nature.comnature.com. We explicitly say "Never" because we don't want future feature creep to try adding some holographic storage or such – it's outside our scope and not needed given the integration with electronic memory.
- **No native Windows or x86 kernel support:** As described, we won't run Windows or legacy OS kernels on bare metal. Some stakeholders might push "can't you just make it run Windows so we don't have to change anything?" The answer is no – doing so would defeat the purpose of our clean-slate design and reintroduce tons of complexity (it'd require an x86 hardware mode, which we refuse to do). We provide other means (VMs) but that's it. This keeps the architecture simpler and avoids legal issues of implementing others' ISAs without license.
- **No fully offline calibration mode:** The system must calibrate while running. In other words, you cannot just calibrate the photonic part once at boot and expect it to run hours or days without recalibration. If someone demands a mode where the photonic tile is "air-gapped" from control during operation (for fear of interference or just misconception), we have to say no; periodic calibration is fundamental, otherwise errors will accumulate. We will design calibration to be as non-intrusive as possible, but it will always be there in the background.
- **No promises of magical performance beyond physics:** We clarify that while photonics can give great speedups for certain operations (matrix multiplies, etc.), the **overall system speedup** for real applications will depend on balancing compute and data movement. The biggest wins likely come from eliminating I/O bottlenecks (hence our emphasis on optical fabric). We caution that raw "TOPS" (tera-operations) figures of photonic MACs can be misleading if the rest of the system can't keep them fed with data. So, we focus on balanced design. If someone in marketing tries to claim "infinite speed" or such nonsense, our engineering stance pushes back with reality. Users should expect big gains on workloads that match well (e.g., large neural network layers, etc.), but not every program will magically be faster (e.g., a program that is 90% branchy logic won't go faster just because there's a photonic unit on the side doing nothing in that case).

In short, by saying these "No" items, we keep the project grounded and avoid chasing rabbits down holes that historically have been unproductive.

29

Key Points

- **Clean-Slate ISA (x147) + Photonics:** Design a new 64-bit ISA with photonic acceleration (OPTX) and an optical interconnect as core features. Don't try to implement x86 in optical hardware – instead, build a new architecture (**x147** with **photonic MAC tiles** and an **optical fabric**), and handle x86 compatibility via dynamic binary translation in software/hardware.
- **CMOS for Logic, Photons for Math:** Keep what CMOS does best (general-purpose logic, control flow, precise state, memory access) in conventional CPU cores. Use photonic hardware for what it does best (fast, parallel linear algebra and high-bandwidth communication). The two work in tandem, managed by a custom kernel optimized for this split.
- **Binary Translation for x86/x64:** Legacy x86 code runs via a combination of JIT translation (for user-mode applications, yielding near-native speed in many cases) and full virtualization (for complete OS environments when needed). No attempt to run x86 instructions directly in hardware beyond what the translator does. This avoids legacy complexity and IP issues, while still allowing users to run their existing software. Published performance ranges and guidelines will set expectations (often fast, but not always 100% of native – rebuild for best results).
- **Photonic Compute Tiles:** Each socket has a photonic accelerator tile (64×64 MZI mesh) integrated via 2.5D. It performs 64-way matrix operations with ~nanosecond-scale latency. Photonic ops are triggered via new OPTX instructions/queues. The design includes continuous calibration loops and on-chip monitoring to manage analog drift, leveraging techniques like chip-in-the-loop optimization to maintain accuracyresearchgate.netresearchgate.net. Photonics deliver major speedups for matrix math, but require careful control – which our hardware and microkernel provide in real-time.
- **Optical Fabric:** Sockets are interconnected with an optical backplane network delivering terabits of bandwidth with low latency. This fabric, managed with credit flow control, allows memory and coherence traffic between CPUs/accelerators at speeds impossible with copper linksbusinesswire.combusinesswire.com. This is critical for scaling workloads across multiple chips without bottlenecking on data movement.
- **Free-Form Memory/Fabric Control Panel:** The system provides a user-visible interface to control and partition memory bandwidth and interconnect resources. Instead of fixed resource scheduling, users can allocate HBM bandwidth shares, set QoS profiles (compute-first, graphics-first, etc.), and reserve optical fabric bandwidth for specific uses. Telemetry is provided for feedback. This ensures that the novel resource (photonic bandwidth, etc.) can be tuned to match workload needs, and that real-time tasks get the needed allocation.
- **Ejectable Secure ROMs:** Removable ROM cartridges serve as secure boot anchors and carry calibration data. The root-of-trust boots from immutable code, then can load trusted firmware from these modulesmicrochip.commicrochip.com. It provides recovery options and easy firmware swaps for maintenance or upgrades, all while using strong cryptographic verification of contents. Essentially, a physical key for the machine's brain – offering both

security (you can pull it out to prevent unauthorized boot) and serviceability (swap in a known-good image if needed).

- **x43 Micro-Hypervisor:** A lightweight kernel that divides the system into real-time and best-effort domains to manage the photonic hardware deterministically. It runs user-space OS services (drivers, etc.) for flexibility. It orchestrates the photonic tasks with microsecond precision and handles the x86 translation environment. Think of it as a custom operating system optimized for a hybrid photonic-electronic computer.
- **Initial Focus on Bandwidth and Linear Algebra:** The first use-cases and benchmarks will target things like large AI model layers, scientific computing kernels, and quantum error correction decoding – scenarios that benefit most from massive linear algebra throughput and fast inter-node communication. By demonstrating big wins in these areas, we make the case for the architecture. Over time, more applications can be optimized to use the photonic path (but we manage expectations that it’s not universal).

In conclusion, this design marries bleeding-edge photonic technology with pragmatic computing architecture. It does so by clearly defining roles: x147 gives us a fresh foundation without legacy cruft, photonics gives us speed in the lanes that matter, and our translation and control systems bridge the old and new worlds. The result (if executed well) will be a system that can run today’s software (with some help under the hood) but also push well beyond today’s performance limits in throughput and latency for tomorrow’s demanding workloads. It’s a bold path, but one grounded in current research and tech trends – and we’ve called out where physics or practicality imposes limits, to ensure we focus on real, achievable gains.

31