The Quest for Unification: A Survey of Hardware-Agnostic Machine Learning Systems

PAUL DUTTON[*], University of Texas at San Antonio, USA

The fragmentation across machine learning system stacks—from high-level programming languages to hardware accelerators—creates significant barriers to research productivity and deployment efficiency. This survey examines how unifying technologies like MLIR, ONNX, and emerging solutions such as Modular's MAX and Mojo aim to address these challenges by providing hardware-agnostic abstractions without sacrificing performance, potentially democratizing access to cutting-edge ML capabilities across diverse hardware platforms.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Additional Key Words and Phrases: Artificial Intelligence, Heterogenous Compute, Compilers, Frameworks, Programming Languages, Edge Compute, Spiking Neural Networks, ASICs, GPUs, TPUs, FPGAs

ACM Reference Format:
Paul Dutton. 2025. The Quest for Unification: A Survey of Hardware-Agnostic Machine Learning Systems. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 20 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The dramatic rise of deep learning has created an unprecedented demand for specialized computing hardware. Modern AI software developers have to work with an increasing number of hardware platforms, software frameworks, and programming paradigms to stay on the cutting edge. The fragmentation often means that developers become tied into a specific ecosystem with difficulties changing these things, even within the same vendor. Expertise becomes siloed, and promising research directions might not get the attention they deserve due to a lack of compatible hardware support.

While General Purpose GPUs (GPGPUs) initially democratized AI research and still serve many well, the development of a newer generation of Application Specific Integrated Circuits (ASICs) for deep learning such as Tensor Processing Units (TPUs), Neural Processing Units (NPUs) which are sometimes referred to collectively as xPUs, along with specialized systems like those from Groq, Cerebras, and Graphcore. Each hardware requires a new instruction set, compiler toolchain, libraries, and programmer training. These hardwares typicall operate alongside a Central Processing

[*]Both authors contributed equally to this research.

Author's Contact Information: Paul Dutton, paul.dutton@my.utsa.edu, University of Texas at San Antonio, San Antonio, Texas, USA.

Unit (CPU) in a heterogenous computing system where tasks are strategically divided among the most capable hardware for that task.

This hardware diversity creates many hardware and software challenges. Python, which is rather slow but easy to learn, is a common developing programming language for a lot of deep learning. Familiar libraries such as TensorFlow, PyTorch are easily accessible in its ecosystem and call to more better performing C++, CUDA, or other faster code underneath. When developers need functionality beyond what is immediately and obviously available from Python, typically the only option is to turn to writing your own C++ / GPU functions and kernels in a language that operates closer to the metal. Few developers possess the expertise in writing Parallel Thread Execution (PTX), Nvidia's assembly language, although it is common to need to do so for the best results.

Rapid prototyping might be done in Python and then for production code a different team entirely will rewrite much of the project in that higher-performance or domain-specific language (DSL); this causes production bottlenecks, new learning curves, and fragmentation.

This situation exemplifies what Hooker terms "The Hardware Lottery" [11]—research directions are disproportionately influenced by available hardware and its accessibility. Many ideas might be theoretically viable, but if they don't align well with Nvidia's CUDA based hardware architecture, the ideas appear dead in the water. As we witness the development of new hardware that differs greatly from modern GPUs in efforts to address some of their shortcomings, the research community becomes more and more fragmented. Fragmentation and competition are indeed joined at the hip, even are we are as a research community largely focusing on the context of transformer based Large Language Models (LLMs) and the kinds of algorithmic operations that make them possible. Alternative paradigms like neuromorphic computing and analog approaches face even steeper adoption barriers as they fundamentally diverge from conventional digital architectures.

As the hardware landscape diversifies, the field needs unifying technologies that can bridge these gaps. This survey examines approaches that aim to provide hardware-agnostic abstractions without sacrificing performance. We will:

1. Analyze current ecosystem fragmentation and its impact on research and deployment 2. Examine model exchange formats and intermediate representations that facilitate cross-platform compatibility 3. Evaluate compiler technologies that abstract hardware complexity from developers 4. Survey the evolving hardware acceleration landscape 5. Assess unifying approaches, with particular focus on ONNX's success in model exchange and Modular's integrated MAX/Mojo stack 6. Discuss implications for future research and development in a heterogeneous computing environment

Through this analysis, we hope to provide insight into how the field might overcome fragmentation barriers to democratize access to cutting-edge ML capabilities across diverse hardware platforms.

2 The Fragmentation Problem

Fragmentation in the AI ecosystem manifests along multiple dimensions: hardware architectures, software frameworks, programming models, and deployment environments. As specialized hardware emerges to accelerate operations needed by newer models, software stacks must be substantially rewritten, often from the compiler level up, in order to leverage these capabilities. This creates a significant lag between hardware launches and their practical usability. Beyond technical barriers, economic and organizational factors further encourage developers to remain within familiar architecture stacks, as migration costs can be prohibitively high in terms of expertise development, code rewriting, and performance optimization.

2.1 The Current AI Ecosystem

The AI software landscape is dominated by a handful of frameworks, each with their own programming models and hardware compatibility profiles. TensorFlow, PyTorch, and more recently JAX provide comprehensive environments for model development, training, and deployment. These frameworks offer high-level abstractions that shield developers from underlying hardware complexity while providing options to introduce handwritten optimizing components.

Despite their capabilities, no single framework universally supports the full spectrum of computing platforms, from mobile processors to specialized ASICs, many of which remain only internally available to the companies that have developed them. Framework compatibility is typically strongest for mainstream hardware (x86 CPUs and NVIDIA GPUs) with varying levels of support for other platforms. This creates a stratified ecosystem where the most accessible tools work primarily with the most dominant hardware:

- **NVIDIA GPUs** are supported by CUDA (introduced 2006), which has evolved into a comprehensive ecosystem of libraries (cuDNN, cuBLAS), profiling tools, and optimization frameworks. These dominate the market [13]
- **AMD GPUs** are accessible through ROCm and HIP (Heterogeneous Interface for Portability), with the latter specifically designed to provide CUDA compatibility in efforts to bring developers over
- **Intel's** ecosystem leverages OneAPI as a unified programming model coinciding with its entry into the discrete GPU market. This Application Programming Interface (API) has brought the support of The Linux Foundation as an open-source direction for the market
- **Edge devices** are targeted through specialized frameworks like TensorFlow Lite, requiring additional model optimization and often complete redeployment pipelines

Performance-sensitive applications frequently require developers to directly access hardware-specific APIs. The financial stakes are substantial—with large-scale models running on thousands of GPUs, even modest efficiency gains can yield significant cost savings. For instance, techniques like kernel fusion implemented in FlashAttention [5] have delivered substantial performance improvements for transformer architectures by optimizing memory access patterns and computation scheduling specifically for GPU execution models.

2.2 Vendor Lock In

NVIDIA has established dominant market position in AI acceleration, with their CUDA ecosystem becoming the de facto standard for high-performance deep learning. This dominance creates several forms of lock-in:

Toolchain Dependencies: NVIDIA's comprehensive ecosystem includes libraries, profilers, and development environments that developers come to rely on[?]. These tools often have specific version interdependencies that complicate migration [?].

Hardware-Specific Optimizations: Organizations frequently invest in developing highly optimized, hardware-specific implementations of performance-critical operations. DeepSeek, for example, achieved superior performance through custom PTX (Parallel Thread Execution) code specifically targeting NVIDIA hardware [6], bypassing CUDA entirely. This investment in hardware-specific knowledge creates significant switching costs.

Architectural Fragmentation: Optimizations frequently target specific generations of hardware within the same vendor's lineup—code optimized for NVIDIA's H100 GPUs may require significant reworking for A100s [?], despite sharing the same fundamental architecture. This issue compounds across vendor boundaries.

Selective Implementation of Standards: While cross-platform standards like OpenCL theoretically enable hardware-agnostic development, their practical utility is limited by incomplete implementations. NVIDIA's OpenCL support, for example, notably excludes access to tensor cores[**?**], specialized units that provide substantial acceleration for matrix operations central to deep learning [**?**]. This strategic limitation ensures that peak performance remains accessible only through vendor-specific programming models. This creates a significant knowledge investment that becomes a sunk cost, reinforcing lock-in even when alternative platforms emerge. Once teams have invested in developing this specialized expertise, the switching costs to another platform become prohibitively high, even if the alternative platform offers theoretical performance advantages.

Proprietary Access Models for New ASICs: Emerging AI-specific hardware like Google's TPUs, Cerebras's Wafer-Scale Engine, Groq's Tensor Streaming Processors (TSPs), and Tenstorrent's Grayskull processors typically operate through proprietary interfaces, often accessible only through cloud services or with significant customer commitments. This creates additional fragmentation as each platform requires distinct programming models and optimization strategies.

[FIGURE HERE FOR SHOWING WHY THESE NEW ASICS ARE DESIRABLE].

This multi-faceted lock-in reinforces "The Hardware Lottery" effect; not only are researchers and developers constrained by available hardware, but they become increasingly committed to specific vendors and architectures through accumulated expertise and optimization investments. Each migration consideration must weigh not just immediate porting costs but also the performance penalties of operating on less-optimized platforms.

3 Model Exchanges and Framework Unification Approaches

To address fragmentation in the AI ecosystem, the field has developed two complementary approaches: standardized model exchange formats that enable cross-framework compatibility, and unified programming frameworks that attempt to provide hardware-agnostic abstractions. These efforts represent critical infrastructure for reducing the switching costs between different hardware and software stacks.

3.1 ONNX as a Standard Exchange Format

It isn't to say that the field is completely void of tools that are more or less standard, available, accessible, and useful. The Open Neural Network Exchange (ONNX) format has emerged as a cornerstone of interoperability in the fragmented AI ecosystem. By providing a standardized representation of computational graphs, operators, and weights, ONNX enables models to move between disparate frameworks and deployment environments.

ONNX achieves this through three key components: (1) a well-defined operator specification covering common neural network operations, (2) an extensible computation graph representation that captures model topology and dependencies, and (3) a standardized serialization and storage format. This allows models trained in one framework to be deployed in another without retraining or substantial redevelopment.

The practical impact of ONNX has been substantial, creating pathways between previously isolated ecosystems. Models developed in PyTorch can be deployed in TensorFlow environments and vice versa, significantly reducing framework lock-in. This interoperability is particularly valuable in deployment scenarios, where hardware and software constraints might differ from training environments.

While ONNX has proven effective for mainstream model architectures, challenges remain for more experimental or specialized designs. Work done [12] has identified several common failure modes, primarily stemming from framework-specific operations, data type incompatibilities, and handling of complex control flow structures. Despite these limitations, ONNX has established itself as a vital "glue" for the industry, enabling practical cross-framework compatibility for the majority of production use cases.

3.2 Hardware Agnostic Framework Approaches

Beyond model exchange, the field has seen numerous attempts to create programming frameworks that abstract away hardware specificity while maintaining performance. These frameworks take diverse approaches to the fundamental tension between hardware-specific optimization and cross-platform compatibility.

3.2.1 Cross Platform APIs: OpenCL and SYCL. Open Computing Language (OpenCL) launched as an early effort to unite parallel workload programming [?] in heterogeneous systems. It does, however, offer targeting of GPUs, CPUs, FPGAs, and some work has even been done to integrate quantum processing into the project for heterogeneous computing [?]. The standard offers a common abstraction model for many use cases.

Despite its ambitious goals and successes in uniting the field in many ways, OpenCL has faced significant practical limitations. Performance often lags behind vendor-specific programming models, and support for advanced hardware features can be incomplete. Notably, NVIDIA's OpenCL implementation lacks support for tensor cores [?], creating a substantial performance gap compared to CUDA-based implementations for deep learning workloads. Furthermore, the collaborative "design by committee" approach has faced criticism for impeding innovation. Apple, an original proponent, eventually abandoned OpenCL in favor of its proprietary Metal framework to better support its custom silicon [?].

Building on lessons from OpenCL, the Khronos Group developed SYCL, which takes an Embedded Domain Specific Language (eDSL) approach. Implemented as a C++ template library, SYCL maintains OpenCL's cross-vendor aspirations while providing a more modern programming model based on C++17 standards. SYCL has gained momentum as a foundation for other cross-platform initiatives, including Intel's OneAPI. OneAPI is offered from Intel as a competing open-source library set for high performance compute on many devices against OpenCL and CUDA. While Intel remains in charge, they offer support for many other devices, from CPUs to GPUs to FPGAs. [2]

3.2.2 Apache TVM. Apache has its TVM, a machine learning compiler for CPUs, GPUs, and other accelerators that makes judicious use of auto-tuning in its optimizing steps to help target a wide range of hardware[4]. Auto-tuning tests parameters for operations to find out what is actually best for that target platform. By removing some of this work from the programmer and automating 'trial and error' testing, the idea is that you should be able to get very good performance on a wide range of targets without manual fiddling. It is also MLIR based, making interoperability between other tools using MLIR more possible through the use of shared dialects.

4 Hardware Acceleration Landscape

The AI hardware ecosystem has rapidly diversified beyond traditional CPUs and GPUs to include specialized accelerators. This diversification creates both opportunities for performance improvements and challenges for creating unified software systems. This section surveys the major hardware platforms driving AI workloads and examines how their architectural differences contribute to ecosystem fragmentation.

4.1 General Purpose Hardware

4.1.1 Central Processing Units (CPUs). CPUs remain the foundation of computing systems, characterized by sophisticated control logic, high single-thread performance, and versatile I/O capabilities. Some begin to incorporate support for Single Instruction Multiple Data (SIMD) extensions like

AVX-512 that can accelerate AI operations, although these are best offloaded to external dedicated devices.

While these can be used for AI directly, they often have few cores to work with which makes parallelization difficult. Instead, they are often the main orchestrating hardware for heterogeneous systems. This includes data management, scheduling, handling workloads with complex logic that don't work well on massively SIMD architectures such as GPUs, and providing a well-known and easily targeted platform for any other such needs.

4.1.2 Graphics Processing Units (GPUs). Originally designed for rendering 3D graphics, GPUs have become the dominant platform for deep learning computation. A screen of many pixels offers a great landscape to develop and exploit parallelism, and GPUs excel in that area. Their architecture features thousands of relatively simple cores organized into compute units that execute the same instruction across multiple data elements, effectively implementing a massive SIMD paradigm. This architecture aligns naturally with the data-parallel nature of neural network operations, where identical computations are performed across large, N-dimensional tensors.

NVIDIA's CUDA platform, introduced in 2006, transformed GPUs from specialized graphics processors to general-purpose computing devices by providing accessible programming models and comprehensive library ecosystems. The resulting General-Purpose GPU (GPGPU) paradigm proved extraordinarily effective for deep learning workloads, which consist primarily of parallelizable matrix multiplications and similar operations.

GPUs also have adapted to this new role, with specialized hardware segments like Tensor Cores, incorporation of higher amounts of High Bandwidth Memory (HBM), interconnection technologies such as NVLink for multi-GPU systems, and support for increasingly common faster data types such as FP16 or INT8[**?**].

4.1.3 Tensor Processing Units (TPUs). Google's Tensor Processing Unit represents one of the earliest purpose-built AI accelerators. First deployed internally in 2015 and later made available through Google Cloud, TPUs feature a systolic array architecture specifically optimized for matrix multiplication operations central to deep learning. This architectural specialization enables higher computational density and energy efficiency compared to general-purpose processors at the cost of their ability to generalize.

TPUs popularized the utilization of the Bfloat16 data format which keeps a wide dynamic range whilst reducing precision. As these chips are intended to work in multi-TPU systems, special attention was given to collective communication operations and memory bandwidth to make large scale systems feasible.

While TPUs offer significant performance advantages for supported workloads, they remain accessible only through Google's cloud infrastructure, with programming models generally coupled to Google's TensorFlow ecosystem.

4.1.4 Neural Processing Units (NPUs). Neural Processing Units have emerged as common accelerators for edge and mobile AI applications. AMD's XDNA architecture, Apple's Neural Engine, and various mobile SoC designs from Qualcomm and MediaTek all incorporate NPUs. These accelerators are typically integrated into System-on-Chip (SoC) designs alongside CPU cores, sharing I/O and memory subsystems.

NPU architectures generally prioritize energy efficiency over raw performance, making them suitable for battery-powered devices. They often feature fixed-function units for common operations (convolution, activation functions) and optimized datapaths for reduced-precision computation. However, their programming models remain highly fragmented, with each vendor providing proprietary software stacks and limited interoperability.

4.1.5 Enterprise Level Accelerators.
The market for data center AI acceleration has expanded dramatically beyond GPUs, with several companies developing specialized hardware for training and serving large models:

Cerebras Wafer-Scale Engine Cerebras takes an extreme approach to scaling, building processors that span an entire silicon wafer [?]. This architecture eliminates traditional chip boundaries, providing massive on-chip memory and computational resources within a single device. While offering unprecedented scale, the architecture requires specialized programming models and cooling infrastructure.

Groq Tensor Streaming Processor Groq's architecture prioritizes deterministic execution through a novel approach to scheduling [1]. Rather than dynamically assigning work to compute units, Groq's compiler generates a precise execution schedule known at compile time. This approach enables predictable performance and latency guarantees particularly valuable for inference workloads.

Graphcore Intelligence Processing Unit (IPU) The IPU architecture emphasizes fine-grained parallelism and in-processor memory to minimize data movement. Unlike GPUs, which execute identical operations across many data elements, IPUs support more flexible execution patterns suited to complex computational graphs with irregular structure.

Intel Gaudi and GNA Intel offers diverse AI acceleration options including Gaudi processors acquired from Habana Labs and the Gaussian and Neural Accelerator (GNA) for low-power speech and audio processing. This portfolio approach provides specialized solutions for different AI deployment scenarios but further contributes to ecosystem complexity.

[FIGURE HERE] shows the performance offered by many of these systems; highlighting the good idea of developing new ASICs for AI workloads. Competition can be good for many actors in the market, but proliferation still has an extraordinary number of hurdles to overcome if we are to see full integration of these new designs into the mainstream workflow, easy access for research and business, and the successes of these projects to leave a lasting impact on direction for the industry as a whole.

4.2 Emerging Hardware Approaches

Beyond conventional digital accelerators, several alternative computing paradigms are being explored for neural network acceleration:

4.2.1 Field-Programmable Gate Arrays (FPGAs).
FPGAs offer reconfigurable hardware that can be optimized for specific neural network architectures. This flexibility enables substantial power efficiency improvements compared to fixed-architecture processors, with studies reporting up to 10x better energy efficiency than GPUs for certain workloads [?]. FPGAs are particularly valuable for edge deployment scenarios with stringent power constraints or rapidly evolving model architectures.

Despite these advantages, FPGAs face significant adoption barriers in AI workflows. Their programming complexity, lengthy synthesis times, and limited software ecosystem support restrict their mainstream use. Projects like ONNX-to-FPGA conversion tools [?] aim to simplify deployment, but significant gaps remain compared to GPU software ecosystems.

4.2.2 In and Near-Memory Computing.
The "memory wall"—the growing disparity between computation and memory access speeds—represents a fundamental bottleneck for neural network performance. In-memory computing architectures address this challenge by performing computations directly within memory arrays, dramatically reducing data movement costs [?].

As models grow faster than our hardware capability and Moore's Law, engineers also look to novel ideas such as switched capacitors for matrix operations, analog computing [?], and computing in-memory or closer to it than we have previously [?] [?].

4.2.3 Neuromorphic Computing. Neuromorphic architectures draw inspiration from biological neural systems, implementing computational models that more closely resemble brain function than conventional artificial neural networks. Spiking Neural Networks (SNNs) represent a prominent neuromorphic approach, using discrete, time-dependent spikes for information transmission rather than continuous activation values [?]. Sure, IBM has their NorthPole and Intel their Loichi line of chips that implement SNNs directly in silicon architecture, but these are inaccessible in innumerable ways (an architecture without a program counter is a difficult target for a compiler), and make the problems in the current landscape look like the mildest of roadblocks. The hardware to run these is vastly different from a GPU; while they are massively parallel, there is no traditional program counter as operation is effectively entirely asynchronous and event driven, and traditional program pipelines will not work.

These architectural differences make neuromorphic systems largely incompatible with mainstream AI software stacks. Most compilers and the tools that make them possible are designed for devices with common architectures (Von Neumann CPUs, Harvard for smaller DSP or microcontroller boards, GPUs being controlled by the CPU in a heterogenous environment). Compiler infrastructures including MLIR lack support for the asynchronous, event-driven computation models that neuromorphic hardware requires. This incompatibility exemplifies an extreme case of "The Hardware Lottery" [11], where potentially valuable approaches remain underexplored due to misalignment with dominant hardware paradigms.

As we scale further into High Performance Computing for large models, the Single Instruction Multiple Data (SIMD) approach of our GPUs moves into Multiple Instructions Multiple Data (MIMD) environments and we see chips planning for this kind of asynchronous operation at the higher layers of abstraction, such as with TPUs, Groq's accelerators, SambaNova, and even HPC graph execution architectures such as the Intel Configurable Spatial Accelerator (CSA).

4.3 Implications for Unification

The diversity of hardware architectures—from conventional CPUs and GPUs to specialized AI accelerators and emerging computing paradigms—presents profound challenges for creating unified software systems. Each platform offers unique performance characteristics and programming models, creating natural fragmentation in the ecosystem.

As we move from the Single Instruction Multiple Data (SIMD) paradigm of GPUs toward more asynchronous Multiple Instruction Multiple Data (MIMD) architectures like those in TPUs, Groq accelerators, and neuromorphic systems, the complexity of unification increases substantially. Even advanced compiler infrastructures like MLIR face fundamental limitations in spanning these architectural divides.

Still, this doesn't mean that we can't begin to plan for diverse hardware and future-proof our AI stacks. Cooperation between hardware designers and software algorithmic specialists can certainly help to make the path forward more clear [?]. Hardware-software co-design approaches that consider compilation pathways alongside architectural innovations may help bridge these gaps. However, as Hooker notes in "The Hardware Lottery" [11], architectural paradigms that lack accessible developer tools and programming models will likely remain underutilized regardless of their theoretical advantages.

5 The Role of Compilers

Compilers play a pivotal role in addressing hardware fragmentation by providing layers of abstraction between high-level programming models and diverse hardware targets. These tools take higher level code, parse it, often apply their own optimizations, and eventually target a hardware platform. Modern compiler infrastructures for AI workloads have evolved dramatically from traditional approaches, incorporating specialized intermediate representations, auto-tuning capabilities, and domain-specific optimizations. This section examines how compiler technologies contribute to unification efforts across the fragmented AI ecosystem.

5.1 Compiler Intermediate Representations

Intermediate Representations (IRs) form the cornerstone of modern compiler design, providing a standardized internal format that decouples front-end languages from back-end code generation. This separation enables a critical advantage: a single IR can support multiple source languages and target multiple hardware platforms, creating a many-to-many mapping that reduces implementation complexity. For instance, if we have some m programming languages, and some n target hardware platforms, without IRs we would need $m * n$ solutions to write any code for any platform. An intermediate reduces this drastically to $m + n$ for scaling.

5.1.1 LLVM: The Foundation of Modern Compiler Infrastructure. A famous example stands in LLVM. This originally stood for “Low Level Virtual Machine” but has since expanded to much more, leaving the name orphaned. This compiler infrastructure established many common patterns now fundamental to hardware-agnostic compilation. Many modern languages compile to this well-defined IR, leaving LLVM to handle a lot of the heavy lifting. The popular programming languages C and C++ (through the Clang compiler), Rust, Zig, Swift, and even parts of CUDA all target LLVM.

LLVM’s architecture offers several key benefits for cross-platform development:

- A language-agnostic IR that preserves program semantics while abstracting language-specific details
- A modular design allowing independent development of front-ends and back-ends
- Sophisticated optimization passes that operate on the IR, benefiting all supported languages
- A flexible target description system for modeling diverse hardware architectures

From there, LLVM can target any number of target devices such as CPUs from the ubiquitous x86 platform, ARM, PowerPC, and so on. This makes language development easy whilst creating a common space for compiler and language developers to work in. As new features come to the IR, they can ripple through the ecosystem.

Despite these advantages, LLVM was primarily designed for conventional CPU targets with limited support for heterogeneous computing environments. Its IR lacks native representations for many operations central to deep learning, such as high-dimensional tensor manipulations, specialized activation functions, and distributed computation patterns. These limitations prompted the development of AI-specific compilation infrastructures.

5.1.2 MLIR: Multi-Level Abstraction for AI Workloads. Chris Lattner, who helped guide the development of LLVM, the Clang compiler, and Swift programming language (amongs many other projects), began a new IR project while working at Google Brain called the Multi Level IR, or MLIR, to help address many existing issues with LLVM and to address modern computing needs, with AI workloads in mind and extensibility to theoretically any challenge, such as quantum computing [?]. By offering multiple levels of abstraction, the hope is to make more features available at many levels for all kinds of developers, unify development pipelines, and offer a more flexible system that can be better adapted to a wider range of hardware, new and future. MLIR dialects "cleanly separate domain-spec[?].

MLIR's dialect architecture defines domain-specific abstractions, operations, and type systems that can be progressively lowered through different levels of representation to offer a group of reusable behavior. This approach offers several advantages for heterogeneous AI compilation:

- Domain-specific dialects can represent high-level operations (tensor contractions, convolutions) with rich semantic information
- Progressive lowering through multiple abstraction levels preserves optimization opportunities
- Common infrastructure reduces implementation duplication across AI frameworks
- Extensibility for new hardware architectures and programming models

The impact of MLIR has been substantial, with adoption across major AI frameworks including TensorFlow, PyTorch, and ONNX. Even NVIDIA has incorporated MLIR into its CUDA compilation stack, suggesting its emergence as a unifying technology for the field. MLIR's flexible architecture provides theoretical support for diverse targets including emerging paradigms like quantum computing, though practical implementations for many specialized architectures remain in early stages. It has been open-sourced and is a part of the LLVM Foundation.

Similarly, the Accelerated Linear Algebra (XLA) open-source compiler wants to be the standard for machine learning workloads. It is built off of its own IR, High-Level Optimizer (HLO); this project was a part of TensorFlow, and now JAX [8]. It currently targets a number of hardware including CPUs, GPUs, and TPUs and increasingly incoporates MLIR. The project has been developed alongside a proprietary version by Google who set the project up to target their own new TPU hardware. Chris Lattner of course was a part of that team for some time and has spoken about its successes and failures [?] and how that story shapes his current works.

Later, OpenXLA launched as an open source version that is framework agnostic and broader in its ability to support tensor-computing hardware from many companies and incorporates even more MLIR into its stack. XLA still struggles to optimize ideally for Neural Networks, although it still performs well in the general case [10].

TVM and XLA

5.2 AI-Specific Compilation Strategies

Beyond intermediate representations, modern AI compilers incorporate sophisticated automated optimization techniques that help bridge the gap between hardware-agnostic programming and hardware-specific performance tuning.

5.2.1 *Auto-Tuning for Performance Portability.*

For AI specific tasks, we see the development of auto-tuning strategies in compilers that are able to automatically generate code for different architectures by optimizing parameters to satisfy some objective function, be it memory accesses or other aspects of performance. Industry standard compilers in non-AI areas such as the GCC compiler or Clang compiler might offer some 150-200 possible flags to be passed along for such work. Auto-tuning reduces the burden on developers to manually tune for every target platform, as this space of optimizations is handled for us at compile time.

However, auto-tuning approaches face fundamental challenges. As with any traditional compiler optimization techniques, there are no guarantees of speedups. The parameter space for optimization grows exponentially with operation complexity, requiring sophisticated search strategies to explore efficiently. Traditional approaches—including exhaustive search, genetic algorithms, and simulated annealing—face trade-offs between tuning time and discovered performance [3]. Recent research has applied machine learning to predict performance rather than measuring it directly, potentially reducing tuning overhead.

Apache TVM exemplifies this approach, incorporating a sophisticated auto-tuning system that can automatically discover optimal implementations for tensor operations across diverse hardware targets [4]. TVM's auto-tuner explores a parameter space for each operation (tile sizes, loop unrolling factors, thread allocation strategies) and measures actual performance to identify optimal configurations.

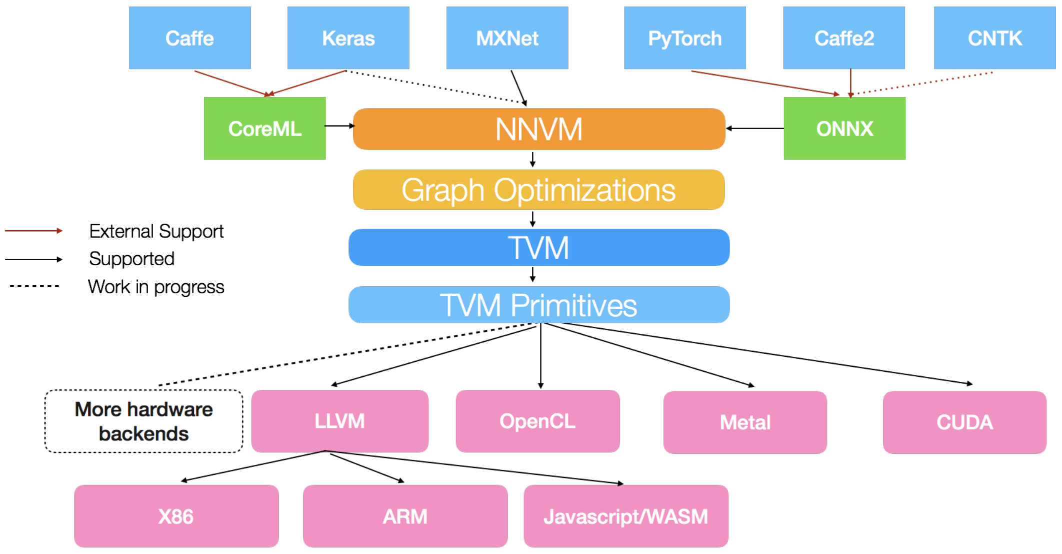

Fig. 1. What a modern ecosystem might look like. Apache TVM. (https://tvm.apache.org/2017/10/06/nnvm-compiler-announcement).

5.2.2 *Heterogeneous Resource Allocation.*

Advanced compilation strategies increasingly consider the entire heterogeneous system rather than individual accelerators in isolation. Projects like ZeRO-Offload demonstrate the potential of intelligent work distribution across CPUs and GPUs, moving appropriate computation and state management to CPUs to reduce GPU memory requirements [?].

This approach has been extended in systems like ZeRO-Infinity, which incorporates NVMe storage into the memory hierarchy for extremely large models [?]. Such systems require sophisticated compiler analysis to determine optimal data placement and movement strategies based on hardware characteristics, including memory capacities, bandwidth limitations, and compute capabilities across the system.

5.3 Domain Specific Languages

Domain-Specific Languages (DSLs) represent another approach to addressing fragmentation by providing specialized programming models tailored to particular domains while abstracting hardware details. For AI workloads, several DSLs have emerged offering different trade-offs between expressiveness, performance, and hardware coverage.

5.3.1 *Standalone DSLs: Triton.*

OpenAI has proffered Triton[?], a Python-like language that allows for efficient GPU programming without dipping into another language for kernels, addressing the "Two Language Problem". Based around its own LLVM "Triton-IR" that can compile to PTX and other targets, it claims performance near CUDA first-party libraries. They suggest that in many situations, performance can be faster than writing CUDA kernels in general practice due to its ease of use that helps offset the years of experience needed to write those top-performing CUDA kernels. Others suggest average performance degradations of about 20% over the best hand-tuned kernels and first party libraries [**?**].

Triton's compiler implements sophisticated optimizations including:

- Automatic memory coalescing for efficient GPU memory access patterns
- Shared memory management without manual synchronization
- Automatic scheduling within GPU Streaming Multiprocessors (SMs)

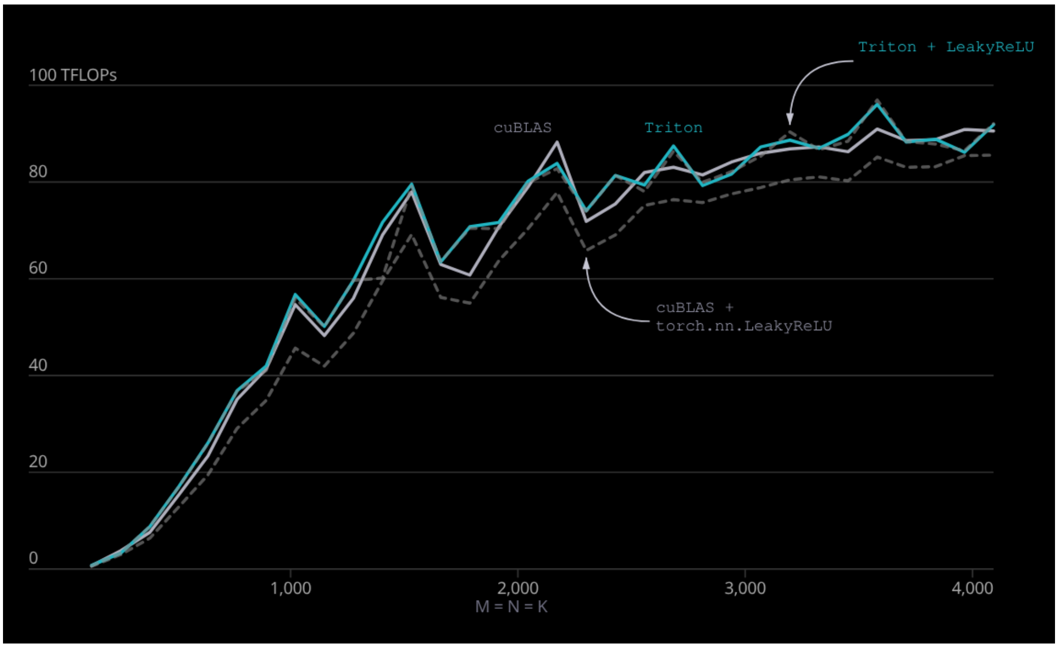

Fig. 2. "V100 tensor-core performance of matrix multiplication with appropriately tuned values for $BLOCK_M$, $BLOCK_N$, $BLOCK_K$, $GROUP_M$." (https://openai.com/index/triton/).

5.3.2 Embedded DSLs for AI. **JAX** stands for the combination of Just-In-Time-Compilation, Autograd (automatic differentiation of functions), and Accelerated Linear Algebra (XLA / OpenXLA). It is a Python library and eDSL that seeks to provide a set of composable function transformations for numerical computing. JAX, coming from Google, leverages XLA as its compilation backend, enabling deployment across CPUs, GPUs, and TPUs without code changes.

SYCL Developed by the Khronos Group, SYCL enables single-source heterogeneous programming in C++. Unlike OpenCL, which requires separate host and device code, SYCL integrates device computation within standard C++ syntax. Intel's Data Parallel C++ (DPC++) extends SYCL with additional features for Intel hardware.

xDSL Built on MLIR's extensible architecture, xDSL provides a framework for creating domain-specific dialects and transformations. This approach enables specialized languages to leverage MLIR's compilation infrastructure while maintaining domain-specific abstractions.

Keras Provides an interface for describing model graphs

Embedded DSLs offer several advantages for unification efforts:

- Familiar syntax reduces the learning curve for domain specialists
- Integration with host language ecosystems provides access to existing tools and libraries
- Compilation to intermediate representations enables hardware-agnostic deployment

However, these approaches often face challenges in error reporting, debugging, and performance predictability compared to standalone languages, as the embedding can obscure the relationship between source code and generated implementation.

5.4 Challenges and Limitations

Despite significant progress, compiler-based approaches to fragmentation face fundamental challenges:

- **Performance vs. Portability:** Hardware-specific optimizations often depend on architectural details that cannot be fully abstracted without performance penalties
- **Representation Gaps:** Existing IRs may lack efficient representations for emerging computation patterns such as sparse operations, asynchronous execution, or quantum algorithms
- **Hardware-Software Co-Evolution:** New hardware architectures often introduce capabilities that existing compiler infrastructures cannot effectively utilize
- **Standardization Complexity:** As with many standardization efforts, the proliferation of "unifying" compiler approaches can ironically increase fragmentation

Other projects exist, such as HTVM's [7] compiler for Edge devices, ONNC [?] for connecting ONNX to accelerators, the HPVM framework and IR [9] and many many more [?]. This list is non-exhaustive. Each attempt to create a universal standard paradoxically risks contributing to further fragmentation. a phenomenon sometimes called the "XKCD 927 Problem" [?], where new standards increase fragmentation rather than consolidate.

These challenges highlight the need for coordinated ecosystem development that considers hardware capabilities, programming models, and compilation infrastructure as an integrated system rather than independent components.

6 Unifying Approaches and Vertical Integration

The preceding sections have examined multiple dimensions of fragmentation in the AI ecosystem and various approaches to address specific aspects of this challenge. Model exchange formats like ONNX address interoperability at deployment time, compiler infrastructures like MLIR provide abstraction layers for hardware targeting, and programming models like JAX offer unified interfaces for heterogeneous computation.

Despite these efforts, no single horizontal unification approach has fully resolved the fundamental tensions between hardware-specific optimization and cross-platform compatibility.

This persistent challenge has led to growing interest in vertical integration approaches that address fragmentation by providing tightly-coupled solutions across the entire stack from programming language to hardware targeting. Rather than attempting to bridge diverse ecosystems horizontally, vertical integration creates cohesive pathways through the stack for specific use cases.

Modular's MAX and Mojo system represents one of the most ambitious recent attempts at vertical integration for AI workloads. By examining this approach in detail, we can better understand both the potential benefits and limitations of comprehensive stack integration as a response to ecosystem fragmentation.

7 Modular's Approach: Vertical Integration

The preceding sections have examined various horizontal unification approaches—model exchange formats, compiler infrastructures, and hardware abstraction layers—each addressing specific aspects of ecosystem fragmentation. While these approaches offer valuable benefits, they operate within the constraints of existing ecosystem boundaries rather than fundamentally reshaping them.

Vertical integration represents an alternative unification strategy: rather than bridging diverse systems, it creates a cohesive technology stack spanning from programming language to hardware targeting. This section examines Modular's MAX and Mojo system as a case study of comprehensive vertical integration in AI infrastructure.

7.1 Vertical Integration Approaches

The persistent challenges of horizontal unification across the ecosystem have prompted interest in vertical integration approaches that connect the entire stack from programming language to hardware targeting. These approaches sacrifice ecosystem breadth for depth, providing tightly integrated solutions for specific scenarios.

Vertical integration offers potential advantages:

- Consistent abstractions across the stack
- Optimization opportunities across layer boundaries
- Unified developer experience without API fragmentation

However, such approaches face significant adoption barriers, requiring developers to commit to a particular stack potentially at the expense of ecosystem compatibility.

7.2 Modular

Chris Lattner has taken his experience from working on various relevant areas (compilers, programming languages, hardware, MLIR, and so on) to found the company, Modular. They offer a totally integrated system starting with their own high-level programming language all the way through an enterprise platform for Kubernetes on the cloud or your own on-premise hardware.

As illustrated in Figure 3, the Modular system comprises three key components:

- **Mojo** - A Python-compatible programming language with systems programming capabilities
- **MAX Engine** - A compiler and runtime system for AI workloads
- **MAX Serve** - Deployment infrastructure for model serving

This architecture incorporates model development, optimization, and deployment within a unified framework, eliminating many of the integration challenges that arise in pieced-together solutions. The system targets deployment across diverse environments including cloud infrastructure and on-premise hardware.

7.3 Mojo Language

The Mojo programming language represents a novel approach to the "two-language problem" faced by AI developers. Rather than accepting the division between a high-level language for model development and low-level languages for performance optimization, Mojo seeks to span this divide within a single language. Mojo is a language built almost as just a front-end to MLIR in order to take advantage of all that it offers. In that way, the Modular team sees the language as a necessary step in their quest to offer a complete and unified platform.

For most functionality, developers will not need to learn anything new. A valid Python program should be a valid Mojo program. The language is different from the common CPython reference implementation of Python in that instead of being interpreted, it's compiled. This means no dipping into C++ for performance, no learning a new language, and a unified developer experience.

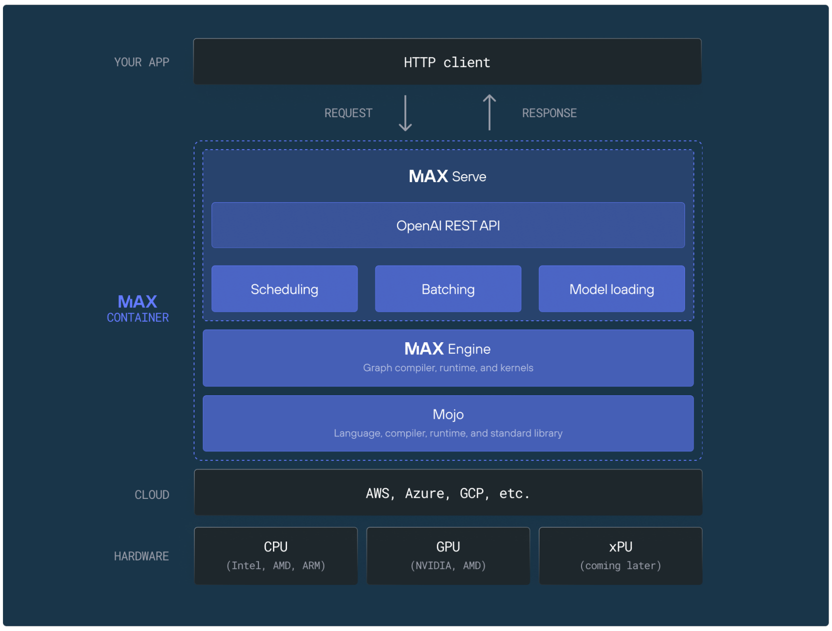

Fig. 3. Complete picture of Modular systems. Modular MAX Github. (https://github.com/modular/max).

7.3.1 Language Design and Python Compatibility. To help entice programmers over even further, any and all existing Python code and libraries can be imported and used as-is without tinkering. The process is painless. You can continue to use PyTorch, JAX, or whatever you've developed without modification. At some point in the future, they hope to bring some of the Mojo improvements to Python to any imported library. This means switching to Mojo should be a no-brainer in most cases. This compatibility enables several key benefits:

- Existing Python libraries can be imported and used directly
- Python developers can incrementally adopt Mojo features
- Organizations can leverage existing Python codebases without rewriting

7.3.2 Systems Programming Capabilities. Beyond Python compatibility, Mojo introduces several features more commonly associated with systems programming languages:

- Static typing with type inference
- An ownership model for memory management inspired by Rust
- Compile-time metaprogramming through parameter specialization, inspired by Zig
- Native SIMD vectorization and parallelization primitives

These features enable performance-critical code to be expressed directly in Mojo rather than requiring implementation in separate C++ or CUDA modules, and they compose to offer powerful compiler optimizations. By exposing systems-level capabilities through a Python-compatible syntax, Mojo aims to make performance optimization accessible to a broader range of developers.

7.3.3 Performance Characteristics. Performance of the language is comparable to the kinds of 'second' languages called by Python traditionally for where performance matters. In comparison to stock Python, the performance gains can be massive. It offers tools for auto-tiling, auto-tuning, compile-time tuning, auto-vectorization, auto-parallelization, and so on. The tools will hopefully continue to mature.

Table 1 illustrates this through a matrix multiplication benchmark comparing various languages and compilers. This is a very naive example of matrix multiplication of square matrices; each bit of code was implemented and tested in very little time to get a feel for accessibility and expressivity. For some tests, we can say B is transposed in $A * B^T = C$, which makes for much more efficient memory access.

The benchmark demonstrates several key insights:

- Unoptimized Python performs orders of magnitude slower than compiled alternatives
- Even optimized Python using NumPy (which calls into C++) remains substantially slower than native implementations
- Mojo achieves superior performance even compared to optimized C with OpenMP parallelization

Of course, the other languages can achieve better performance through use of Basic Linear Algebra Sublibraries (BLAS) libraries and the like, but this stands to show that Mojo gets you up and running with great performance and very little effort.

Code is available at https://github.com/TheAgaveFairy/MatMulComparison, the test platform is an AMD Ryzen 7 7600X CPU with DDR5 and Ubuntu 24.04.

Table 1. Comparison of Matrix Multiplications Across Languages / Compilers

Language / Compiler	Notes	Run Time us
C / GCC	Naive	3,075,132
C / GCC	-O3, Transposed	142,836
C / GCC / OpenMP	-O3, Transposed	28,612
C++ / Zig Clang	2D Vector Naive	19,486,130
Go 1.23.6	Naive	2,168,467
Python 3.12.8	Naive	52,559,288
Python 3.12.8	Numpy (C++), Transposed	1,196,449
Mojo 25.1.0	-O3 (default), Transposed	18,018

7.4 MAX Engine and Serve: Unified Compilation and Deployment

While Mojo addresses programming model fragmentation, the Modular Accelerated Xecution (MAX) Engine and Serve components target compilation and deployment challenges in the AI ecosystem.

The MAX framework offers tools and libraries for model, kernel, and hardware developers. As a part of this, it offers a graph compiler, a runtime, and kernels. It also provides serving tools within containers for pipelining, scheduling, batching, and model loading. MAX wants to be a one-stop shop with the added benefit of having all of these under one roof.

7.4.1 MAX Engine Architecture. By building on MLIR, MAX Engine can leverage the dialect system to represent operations at multiple levels of abstraction. This enables both hardware-independent optimizations and hardware-specific code generation from a common representation. MAX Engine provides a unified compilation and runtime system for AI workloads across diverse hardware targets. Its architecture includes:

- A graph compiler that optimizes high-level model representations

- Specialized kernel implementations for performance-critical operations
- A runtime system that manages execution across heterogeneous devices

A key architectural advantage is the direct integration between Mojo and MAX Engine. Rather than passing through multiple abstraction layers and conversion steps, Mojo code can be directly lowered to efficient implementation through MLIR, potentially preserving more semantic information for optimization.

7.4.2 Hardware Targeting Strategy. MAX Engine currently supports CPUs and NVIDIA GPUs, with AMD support under development. Notably, the system compiles directly to native code for each platform, generating PTX for NVIDIA GPUs without requiring the full NVIDIA toolkit or CUDA runtime. This approach offers several advantages:

- Reduced deployment complexity and container size
- Elimination of version compatibility issues between CUDA components
- Potential for more comprehensive cross-compilation

This direct compilation strategy contrasts with approaches that rely on vendor-provided libraries and runtime systems, potentially reducing dependency on specific vendor ecosystems and their issues [?].

7.4.3 Deployment Infrastructure. Early performance evaluations of the Modular stack show promising results compared to established solutions. Figures ?? and ?? compare MAX Engine against vLLM, a widely-used library for LLM serving, across several benchmark workloads.

These comparisons demonstrate that MAX Engine achieves competitive performance despite lacking specific optimizations like large batch support and paged attention[?] that have been refined in vLLM. The performance metrics show particular strengths in GPU utilization and reduced non-kernel time, suggesting effective system-level optimization. These tools are still early in development.

It is worth mentioning the additional cluster system which aims to achieve large scale, multi-node systems 3. Of course, not only do we have to develop models, but we have to deploy them to customers as well. Offering this extra step could prove beneficial for developer ease.

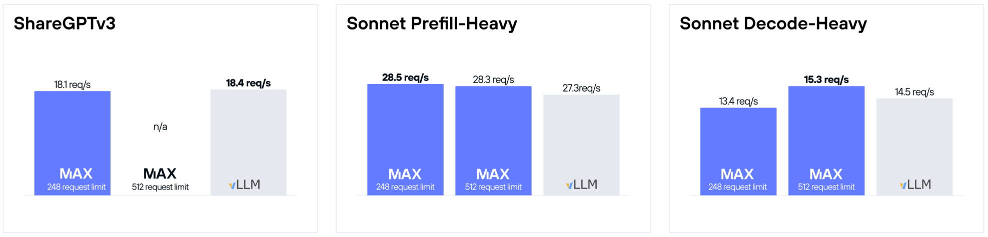

Fig. 4. Comparison of MAX Engine vs. vLLM. Modular Blog. (https://www.modular.com/blog/max-gpu-state-of-the-art-throughput-on-a-new-genai-platform).

7.5 Analysis: Trade-offs

Modular's approach is no panacea, and does offer some drawbacks. Vertical integration can cause new kinds of fragmentation, lose flexibility, and fail in "eggs all in one basket" kinds of ways.

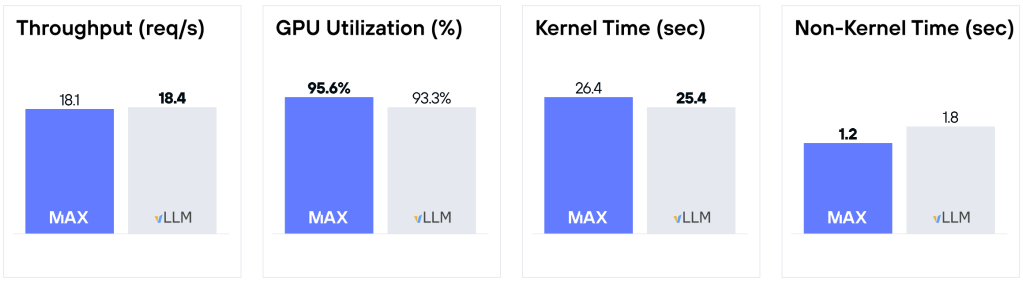

Fig. 5. Comparison of MAX Engine vs. vLLM. Modular Blog. (https://www.modular.com/blog/max-gpu-state-of-the-art-throughput-on-a-new-genai-platform).

7.5.1 *Vertical Integration.* Modular's approach illustrates both the potential benefits and inherent limitations of vertical integration as a unification strategy. The integrated stack offers several advantages:

- Elimination of abstraction gaps between components
- Unified optimization across traditional boundaries
- Consistent developer experience from programming to deployment
- Reduced dependency on multiple vendor ecosystems

However, this approach also involves significant trade-offs:

- A new system still needs to be learned and adopted
- Potentially decreased flexibility to incorporate best-of-breed components from different vendors
- Increased commitment to a single technology stack
- Potential for feature lag compared to specialized solutions in rapidly evolving domains

The fundamental question for vertical integration approaches is whether the benefits of cohesion outweigh the costs of decreased interoperability with the broader ecosystem. For organizations struggling with the complexity of integrating fragmented components, a unified stack may offer substantial advantages despite these trade-offs. If nothing else, this undertaking should offer the industry great insights by providing a new radical paradigm.

The system's reliance on open standards like MLIR provides potential interoperability pathways with other ecosystem components, which may mitigate some of the limitations of vertical integration. This hybrid approach—unified internal architecture with standardized external interfaces—could represent a promising direction for addressing fragmentation while maintaining ecosystem connections.

7.5.2 *One Complicated Language.* Python offers many features and high level abstractions, making the feature set of Mojo already higher than simpler languages. Mojo adds functionality in order to tap into some key features of their MLIR based compiler, although this further increases complexity.

- An ownership and lifetime system like Rust and Swift can be difficult to learn
- The addition of the $fn()$ keyword along with $def()$ for function declaration offers different behaviors to choose between
- Names (variables) may now only have one type, designated by that name's first assignment
- Separation of compile time **parameters** from runtime **arguments** allows for powerful metaprogramming, at the cost of additional syntax and complexity

- Extensive use of function declarations for functionality
- MAX Graph system is a new way to declare operations, marking a new transition for those used to languages like CUDA or Triton

While it may be that one only needs to learn a single language to reach all levels of performance, it is one language that has had to combine both the highest level abstractions of languages like Python as well as the low-level systems level control of C++ and Rust along with its own new ecosystem of MAX tools. Switching to this system will take work from both camps and the alleged benefit of having just one language becomes less obvious as the learning curve can be massive to leverage both ends of the spectrum provided.

8 Conclusion

There is a continuing challenge of hardware and software codesign. Much of the burden lands on compilers, and tools like MLIR certainly are going a long way in standardizing the dialects that everybody will "speak" as the tools are built to get our models running.

Layers of abstraction can make things very easy, but we still must consider the cost of performance for the sake of portability. The tradeoffs weigh differently for each party; research might need to move fast whilst an inference platform might want to fight for every last bit of performance they can to save dollars in their massive operations.

The development and emergence of new standards such as ONNX show important pieces of glue for the industry, and competition between frameworks offers a diverse approach to tackling big problems.

Unifying approaches such as a top-to-bottom rebuilding of our entire stacks from the ground up as seen in Modular are extraordinarily large tasks. If performance can't match other solutions, if compatibility can't be built deeply with other systems, if people don't want to learn yet another system, the project could be dead before it even starts. A one-size-fits all solution might be the best for nobody, as we need custom solutions to compete.

References

[1] Dennis Abts, John Kim, Garrin Kimmell, Matthew Boyd, Kris Kang, Sahil Parmar, Andrew Ling, Andrew Bitar, Ibrahim Ahmed, and Jonathan Ross. 2022. The Groq Software-defined Scale-out Tensor Streaming Multiprocessor : From chips-to-systems architectural overview. In *2022 IEEE Hot Chips 34 Symposium (HCS)*. 1–69. doi:10.1109/HCS55958.2022.9895630

[2] Silvia Alcaraz, Ruben Laso, Oscar Lorenzo, David Vilariño, Tomas Pena, and Francisco Rivera. 2024. Assessing Intel OneAPI capabilities and cloud-performance for heterogeneous computing. *The Journal of Supercomputing* 80 (03 2024), 1–22. doi:10.1007/s11227-024-05958-5

[3] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning using Machine Learning. *arXiv.org* (2018).

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. *arXiv.org* (2018).

[5] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *arXiv.org* (2022).

[6] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[7] Josse Van Delm, Maarten Vandersteegen, Alessio Burrello, Giuseppe Maria Sarda, Francesco Conti, Daniele Jahier Pagliari, Luca Benini, and Marian Verhelst. 2024. HTVM: Efficient Neural Network Deployment On Heterogeneous TinyML Platforms. *arXiv.org* (2024).

[8] Nestor Demeure, Theodore Kisner, Reijo Keskitalo, Rollin Thomas, Julian Borrill, and Wahid Bhimji. 2023. High-level GPU code: a case study examining JAX and OpenMP. In *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. ACM, New York, NY, USA, 1105–1113.

[9] Adel Ejjeh, Aaron Councilman, Akash Kothari, Maria Kotsifakou, Leon Medvinsky, Abdul Rafae Noor, Hashim Sharif, Yifan Zhao, Sarita Adve, Sasa Misailovic, and Vikram Adve. 2022. HPVM: Hardware-Agnostic Programming for

Heterogeneous Parallel Systems. *IEEE MICRO* 42, 5 (2022), 108–117.

[10] Xuzhen He. 2023. Accelerated linear algebra compiler for computationally efficient numerical models: Success and potential area of improvement. *PloS one* 18, 2 (2023), e0282265–e0282265.

[11] Sara Hooker. 2020. The Hardware Lottery. *arXiv.org* (2020).

[12] Purvish Jajal, Wenxin Jiang, Arav Tewari, Erik Kocinare, Joseph Woo, Anusha Sarraf, Yung-Hsiang Lu, George K. Thiruvathukal, and James C. Davis. 2024. Interoperability in Deep Learning: A User Survey and Failure Analysis of ONNX Model Converters. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1466–1478. doi:10.1145/3650212.3680374

[13] Nauman khan. 2024.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009