POLYMORPHIC ARCHITECTURE DEFINITION

OCP FTI AI Co-Design Workgroup

Contributors (in alphabetical order):

Boon Chong Ang	Intel
Kevin Cameron	Cameron EDA
Praveen Gunasekaran	Visa
Sakul Gupta	Micron Technology
Jack Harwood	Marvell Technology
Brian Hirano	Micron Technology
Jai Kumar	Broadcom
Victor Lu	Independent
Danny Moore	AMD
Igor Muskatblit	Independent
Viswateja Nemani	Achronix Semiconductor
Bijan Nowroozi	OCP
Mircea Stan	University of Virginia
George Williams	Independent
James Wong	Palo Alto Electron
Weifeng Zhang	Lenovo Research

Date: September 2024

Executive Summary

As artificial intelligence (AI) continues to evolve at a rapid pace, computing infrastructure supporting it must adapt to meet dramatically ever-changing demands. Innovation in AI models, and the capability to train and infer from these models are constrained by compute platform capabilities. As development will eventually give way to deployment, optimization of these systems for performance and operational efficiency will be imperative. System hardware design cycles are long and design goals are simplified in this phase of reducing development cycles to meet the market with actual platforms. Today's systems do not offer high levels of performance and flexibility while meeting performance and operational efficiency targets. In 2023, a vision addressing this shortcoming was outlined in a whitepaper titled "Polymorphic Architecture for Future AI Applications." It proposed addressing these challenges using hardware software co-design principles. Some key requirements included large-scope scalability, and polymorphic computing infrastructure with hierarchical transformability and composability.

Development of these ideas continued through deeper discussions of architecture, system design considerations, evolution of the State of the Art in AI models and performance expectations. The results of Open Compute Project's (OCP) Community based effort culminates into this new whitepaper which describes, in greater detail, a **Polymorphic Architecture** for AI computing that unleashes unprecedented flexibility, efficiency, and scalability.

Developed by the OCP's AI Co-Design Workgroup, Polymorphic Architecture represents a paradigm shift in how we conceptualize and implement AI computing systems. At its core are two key principles:

1. Continuous transformability
2. Recursive composability

These principles enable computing resources—from individual chiplets to entire data centers—to dynamically reconfigure themselves, optimizing performance for diverse AI workloads in real-time.

Key benefits of Polymorphic Architecture include:

- Seamless scaling from edge devices to massive cloud deployments
- Improved resource utilization and energy efficiency
- Future-proofing against rapidly evolving AI algorithms and models
- Potential for significant cost savings in AI infrastructure

Date: September 2024

- Means to realistically utilize heterogeneous components in systems

This white paper outlines the foundational concepts, technical specifications, and potential applications of Polymorphic Architecture focused on AI computation. It also provides a high level roadmap for implementation, including simulation frameworks and APIs for standardization.

As the AI landscape continues to evolve, Polymorphic Architecture offers a forward-thinking solution to the challenges of next-generation AI computing. Industry leaders, researchers, and developers are invited to explore this transformative approach and contribute to shaping the future of AI infrastructure.

Table of Contents

1. Introduction **8**
Background and Concept 8
Key Characteristics of Fractal Computing 8
Polymorphic Architecture: Enhancing Execution Efficiency 8
Comparison with Evolvable Computing 8
Goals of This Specification 9
Feasibility Assessment and Community Effort 9
Glossary 9
2. Compliance with Open Compute Project Tenets **10**
2.1. Openness 10
2.2. Efficiency 10
2.3. Impact 10
2.4. Scale 10
2.5. Sustainability 10
3. AI Polymorphic Architecture Specification **11**
3.1. Underlying principles / assumptions 11
3.2. Polymorphic Architecture execution flow 12
3.3. Examples of key features 13
3.3.1. Naive composition of sigmoid operator 13
3.3.2. Large language model inference 14
3.3.2.1. Application 14
3.3.2.2. Profiler (Characterization) 15
3.3.2.3. Resource Manager 16
3.3.2.4. Compose / Decompose 16
3.3.2.5. Free 17
4. High Level Architecture Definition **18**
4.1. Definitions of compute capabilities 18
4.1.1. Compute operators and interconnect types 18
4.1.1.1. Compute resource description and APIs 19
4.1.1.2. An example of compute resource allocation 20
4.2. Scale-up of compute units 21
4.2.1. Scale-up description and APIs 22
4.2.2. An example of compute scale-up with composition 22
4.3. Scale-out of compute for larger systems 24

Date: September 2024

4.3.1. Connecting Scale-up and Scale-out to build larger systems 25
4.4. Security Considerations 29
5. Memory Architecture Definition 30
5.1. Definition of basic memory units 30
5.2. Compute Operators and Interconnect Types 31
5.3. Memory hierarchy and composability 31
5.3.1. Compute Resource Description and APIs 34
5.3.2. Example: Compute Resource Allocation 35
5.4. Scale-up of Compute Units 36
5.4.1. Scale-up Description and APIs 36
5.4.2. Example: Compute Scale-up with Composition 36
5.4.3. Scale-out of Compute for Larger Systems 36
5.4.4. Connecting Scale-up and Scale-out 37
5.5. Security Considerations 37
5.6. Memory tiering 37
6. Software Architecture 38
6.1. Basic primitives 38
6.1.1. Numeric data types 38
6.1.2. Aggregate types 39
6.2. Characterizing AI applications 40
6.2.1. Compute graph 40
6.2.1.1. Abstract compute graph 40
6.2.1.2. Graph lowering 41
6.2.1.3. Operators 42
6.2.2. Model characterization 42
6.2.2.1. AI application states 42
6.2.2.2. Inference 43
6.2.2.3. Training 43
6.2.2.4. Fine-Tuning 44
6.3. Computing resource composition 44
6.3.1. Fine-grained instruction sets 44
6.3.2. Coarse-grained instruction sets 45
6.3.3. Heterogeneous Systems 45
6.3.4. Computing topology auto-detection and transformation 46
6.4. Computation partitioning and scheduling 48
6.4.1. Deep Learning Parallelisms 49
6.4.1.1. Data Parallelism 49
6.4.1.2. Pipeline Parallelism 49
6.4.1.3. Tensor Parallelism 50

6.4.2. Collective communication primitives 51
7. Architecture APIs for Simulation 53
7.1. Preliminary high-level APIs for simulation 54
7.2. Reference implementation of simulation 55
7.3. ASTRA-Sim plug-ins 55
8. Conclusion 57
9. Glossary 59
10. References 61
11. License 63
12. About Open Compute Foundation 65
Appendix A. Use Cases 66
1. Inference 66
1.1. Requirement 66
1.2. No Parallelism 66
1.2.1. General Discussion 70
1.2.2. Job Scheduling 70
1.3. Pipeline Parallelism 71
1.3.1. Job Scheduling 72
2. Pre-Training 73
3. Tuning 74
Appendix B. Compute Graph Operations 74
1. Fixed Point Types 85
2. Floating Point Types 86
Appendix C. Security and Threats 88
Appendix D. A sample implementation of PolyArch 90
1. Definitions of key software elements for simulation 90
1.1. Terms borrowed from Google Pathways [3] 91
1.2. Additional System Considerations 93
2. PolyArch operator language specification 93
3. Using the System 93
4. Things not covered/Questions 97
4.1. Checkpointing 98
Appendix E. Topics for future discussions 99
1. Whole-stack view 99
2. Markup language for Polymorphic Architecture data definitions 100
3. Questions for on-going efforts inside OCP 100
4. Analog AI 102

5. Spiking Neural Networks 102
6. Optical Computing 103
7. Quantum Computing 103

1. Introduction

Background and Concept

Polymorphic Architecture represents a novel implementation of a future-oriented AI computing paradigm, proposed by the OCP AI Co-design Workstream. This approach aims to address both spatial and temporal scale-out challenges in AI computation power evolution efficiently. The cornerstone of this concept is **fractal computing infrastructure**, characterized by **continuous transformability** and **composability across hierarchies** of computing resources.

Key Characteristics of Fractal Computing

1. Recursive Decomposition and Recomposition:
    - Computing resources can be recursively decomposed into finer-grained elements at lower hierarchies.
    - These elements can transform and adapt to running applications, enhancing execution efficiency or supporting new functions.
    - Conversely, multiple resource units can be recomposed at higher hierarchies to form larger logical elements for coarse-grained workloads.
2. Native Composability:
    - Facilitates integration of advanced computing capabilities.
    - Incorporates advancements in interconnect technology and architecture development.
    - Supports heterogeneous computing, domain-specific acceleration, architecture disaggregation, chiplets, composable memory, and modular architecture design.

Polymorphic Architecture: Enhancing Execution Efficiency

Building upon fractal computing, Polymorphic Architecture achieves higher execution efficiency through temporal and spatial polymorphisms. It enables:

- Sharing, recomposition, and transformation of underlying compute/interconnect resources.
- Operations at various granularities across vertical hierarchies (from chiplets to racks and beyond).
- Alignment with the growing diversity and rapid evolution of AI applications.

Comparison with Evolvable Computing

While evolvable computing architecture [27] focuses on extensibility for scalable addition of components, functions, or materials over time, Polymorphic Architecture goes further:

- It makes the current architecture both physically (resource) and logically (function) composable.

- It redefines and transforms resources with composability, enabling novel architectural capabilities for execution efficiency.

Goals of This Specification

This architecture specification aims to:

1. Define high-level architecture functions and interconnections at chiplet, chip, and system levels.
2. Outline microarchitecture breakdowns to implement high-level functions.
3. Specify reconfiguration and dynamic partitioning mechanisms for compute and interconnect resources to support polymorphism.
4. Establish the architecture programming model and software framework for simulation, including:
    - Domain-specific language (DSL)
    - Compiler prototyping
    - Software characterization description
    - Control flow of architecture recomposition
    - Compute mapping and scheduling

Feasibility Assessment and Community Effort

The feasibility of the Polymorphic Architecture will be assessed through detailed simulation and experiments. This specification outlines the structure of APIs to enable cycle-approximate simulation. We call for a joint community effort to standardize architectural simulation APIs, fostering the development of a polymorphic computing ecosystem.

Glossary

For definitions of technical terms used in this document, please refer to the glossary in Appendix A.

2. Compliance with Open Compute Project Tenets

2.1. Openness

This AI Polymorphic Architecture is proposed prior to implementation in the open, and is ultimately an extensible architecture. Future contributors are encouraged to modify and extend this vision as necessary.

2.2. Efficiency

Dynamic resource allocation and composability allow for high levels of resource management, thus avoiding stranded capacity issues.

2.3. Impact

Implementation of this AI Polymorphic Architecture would result in a large shift towards commonality with AI stacks for both inference and training. As this proposal is intended, the potential for benefits across the IT ecosystem should encourage industry to align around this approach.

2.4. Scale

Presently there are growing concerns with the lack of consistency in AI stacks across platforms resulting in a lack of a unified approach to load matching and management to platforms. This approach could be utilized uniformly across the IT ecosystem (supply chain) regardless of ISA, networking capabilities, platform configuration or other architectural concerns.

2.5. Sustainability

Ultimately the proposed resource management will increase overall system performance and efficiency resulting in reduced demand for resources such as power, capital equipment, cooling and more.

3. AI Polymorphic Architecture Specification

Hierarchical Abstraction

Figure 1 illustrates the abstraction of Polymorphic Architecture with hierarchical descriptions. At each layer, all architecture components—including hosting/management controller, compute, memory, and interconnect—are uniformly described as resources using definitions outlined in subsequent chapters.

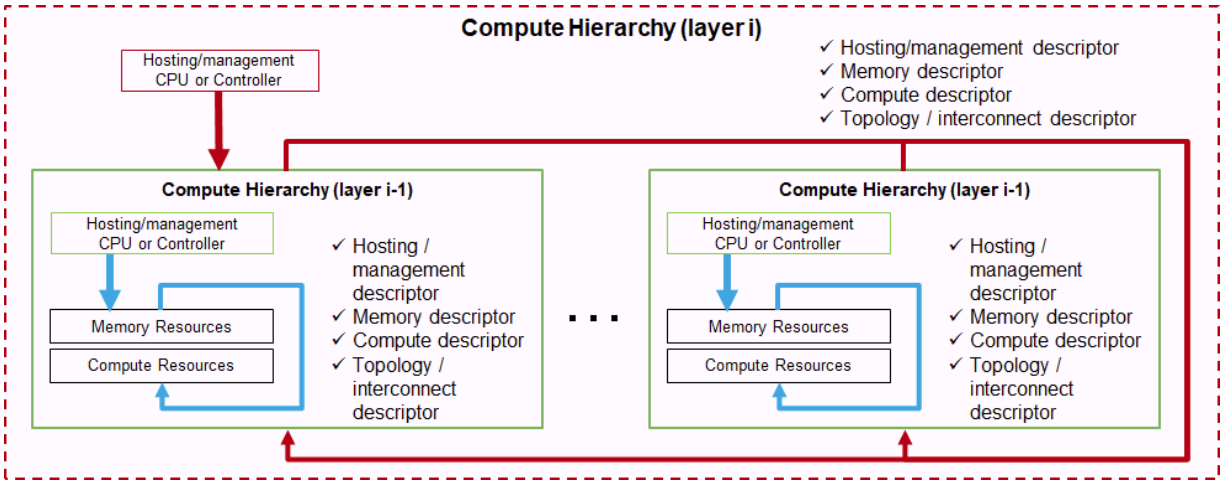


Figure 1. Abstraction of Polymorphic Architecture

In Figure 1, the compute capacity of Layer (i) is described as a hosting (or management) controller with an array of compute units, interconnected with a specified topology. This layer also includes memory and interconnect resources, described by their respective descriptors. Each compute unit can be further described with the same structure in Layer (i-1), enabling recurrent composition and decomposition of compute architecture in this polymorphic paradigm.

This future-oriented design principle positions Polymorphic Architecture as a foundation for accelerating various domain-specific applications.

3.1. Underlying principles / assumptions

- **Ultra-low latency interconnects:** Modern computer architectures can be abstracted as compute units with various interconnect topologies. For large-scale AI applications like LLMs, overall data movement latency is dominated by interconnect bandwidth, rendering individual link latency insignificant ("near-zero latency"). This principle enables the

application of architectural design ideas across various vertical hierarchies, from chip level to large-scale datacenters.

- **Dynamic reconfigurability**: Future green data centers will prioritize resource utilization (goodput and time to completion) as a major driver for architectural innovation. Full configurability reduces stranded resources, boosting application performance and cost savings.
- **Coarse-grained shareable resources**: The architecture balances compute flexibility (general-purpose) and performance (native and domain-specific). It leverages heterogeneous, domain-specific sub-compute-units for better performance while enabling their sharing via architectural re-composition to support non-native or emerging operations.

3.2. Polymorphic Architecture execution flow

Figure 2 depicts a potential execution flow for applications on the Polymorphic Architecture..

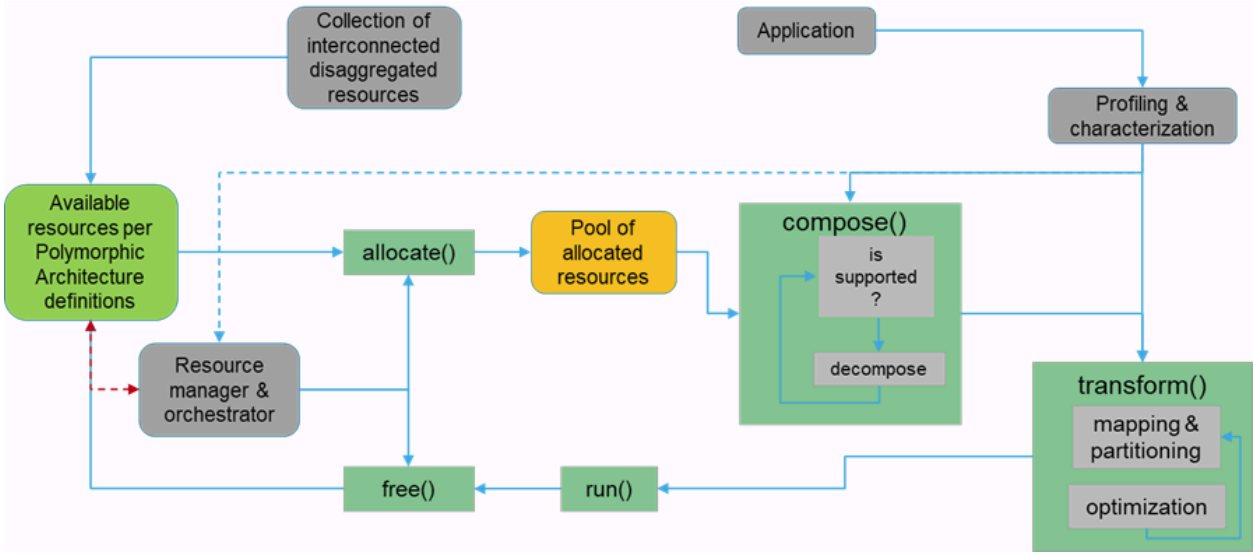


Figure 2. Execution flow of Polymorphic Architecture

A collection of disaggregated compute resources is made available to users under the resource manager. All resources, along with their attributes (capabilities and how they are physically interconnected), are registered to the resource manager based on the uniform descriptions defined by the Polymorphic Architecture. The typical execution flow for AI applications to run on Polymorphic Architecture is as follows:

- The application is profiled and characterized offline to identify what types of compute resources are needed.
- A pool of resources and interconnects are **allocated** by the resource manager according to the application's characteristics.
- The resource pool is **re-composed** and configured at run-time to guarantee that the specified application can safely run on the pool.
- Before the application is launched to **run**, the resources can auto **transform** themselves to adapt to the application's partitioning and optimization strategies.
- The resources are **freed** and made available after the application finishes its execution.

3.3. Examples of key features

Here, we provide a couple of examples to illustrate the key features of polymorphic architecture.

- Naive composition of base compute elements to support Sigmoid operator
- Large language model (LLM) inference

3.3.1. Naive composition of sigmoid operator

This is the definition of sigmoid function. The sigmoid function, expressible as a combination of sub-functions (exp, add, div), exemplifies the composition principle. In a heterogeneous accelerator system, some accelerators may natively support sigmoid, while others don't. Also worth noting that its derivative can be calculated from sigmoid itself.

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

A compute system is composed of multiple heterogeneous accelerators, each of which may support a different set of instructions (or operations). When a sigmoid function in AI models is mapped to this architecture for execution, some accelerators can natively support it, but others may not.

As shown in Figure 3, Accelerator_0, Accelerator_1, and Accelerator_2 all natively support Sigmoid Operator, but Accelerator_3 doesn't. Accelerator_3, composed of chiplets supporting exp, sum, and div operations, can be logically re-composed to run sigmoid efficiently, mirroring native sigmoid accelerators.

It's important to reiterate that in this representative example, a new logical processor is formed to run sigmoid just as efficiently as a native sigmoid accelerator.

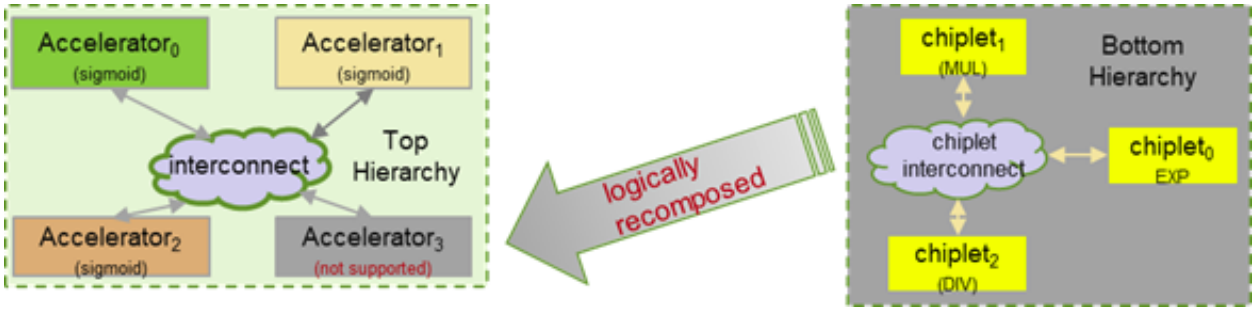


Figure 3. An example of Sigmoid composition using basic arithmetic operators

3.3.2. Large language model inference

This use case, becoming more prevalent, demonstrates scaling LLM model inference in a mesh compute infrastructure.

3.3.2.1. Application

The spectrum of polymorphic applications should support existing methods of defining LLMs. As of this writing, this typically involves leveraging a "composibility" AI framework such as TensorFlow, PyTorch, or JAX to load an existing model from a model repository such as HuggingFace.

As LLMs gain broad traction in the enterprise, we will see a larger variety of LLM applications based on other up-and-coming AI frameworks (consider Nvidia's TensorRT-based Nemo or OpenAI's Triton) and even low or no-code platforms that could see even broader adoption than apps based on AI frameworks. The low-code/no-code applications will be highly dependent on loading pre-trained branded "celebrity" models (Llama, Mistral, Gemini, etc.) weights from secure model repositories.

Additionally, enterprises may decide to focus on LLM support provided by a public cloud provider (AWS, Azure, GCP) because the enterprise data is already stored with that provider. This means that enterprises may stick with specific families of LLM optimized by a provider (GPT in Azure, Llama in Meta, Gemini in Google, Anthropic/Claude in AWS). The landscape is likely to change, but the polymorphic architecture attempts to be agnostic to these industry alignments when possible. That said, it would be too challenging to ignore the LLM "brands"

since a particular family of models might be preferred by applications and the cloud infrastructure where the application is hosted. Therefore the polymorphic architecture supports these model families as a first-class citizen at the application layer.

In addition to model families, applications can request other common LLM model parameters in the application:

- model family: for example “LLama2”
- model size: for example “7B”
- context window size: for example, 32k tokens
- quantization strategy: for example, “8-bit weights”
- batch size: for example, “10”
- token rate: for example, “500 tokens/second”
- accuracy: for example, “90% accuracy on HellaSwag”

The polymorphic architecture supports applications that require specific values of these parameters (for example, “Llama7B, 32k context, 8-bit weights) identifies a specific model that can be downloaded from a model repository such as HuggingFace. The polymorphic architecture should also support application use cases where some or even all parameters are parameters and will make a best effort to resolve to the right model that aligns with the parameters. For example, if the application requires “Llama2” but no model or context size preferred, the polymorphic architecture could choose to start with the largest model in the family and largest context size. Of course, the system may not have the resources to use such a model and must then locate a different model in the family that can be supported by the system. In polymorphic architecture, the “profiler” must have the ability to deal with all of these use cases and will need to negotiate with the part of the system which keeps up to date with resource usage, such as the “resource manager.”

In summary, this polymorphic architecture supports:

- ☐ Various AI frameworks (e.g., TensorFlow, PyTorch, JAX)
- ☐ Low/no-code platforms
- ☐ Cloud provider-specific LLM families
- ☐ Flexible model parameter specifications (e.g., model family, size, context window, quantization strategy)

3.3.2.2. Profiler (Characterization)

As described above, the application has decided on all, some, or no decisions about its LLM model requirements. The “profiler” uncovers these “parameter” requirements, and importantly

must resolve the requirements to a specific model in a model family. While it does this it will also require visibility to what the system can support, which may be dynamic. In a way, the “profiler” performs a type of conflict resolution similar to software package management systems that automatically detect and resolve software version issues during software upgrades.

In a dynamic, multi-user system, the availability of resources (for example, DRAM attached to a compute node) may change quickly over time. This means that the “profiler” not only needs visibility of resource usage in real time, but also the ability to transactionally reserve those resources. In this way,when the model is actually loaded later, the resources have already been reserved and ready for the model load. The “profiler” stage must work closely with the “resource manager” to accomplish this.

The “profiler” will need several core capabilities:

- “inspect” applications to surface parameter requirements
- retrieve specific models from trusted model repositories
- translate static models into run-time (high-level) compute graphs
- lower the high-level compute graph into an intermediate representation of the model specific to available hardware accelerator in the infrastructure

3.3.2.3. Resource Manager

Tracks and allocates:

- Compute nodes (CPU, GPU, xPU)
- Memory (DRAM, SSD)
- Mesh connectivity

The resource manager may incorporate complex decision-making capabilities for optimal resource allocation.

The “profiler” makes the best decision possible about what resources need to be reserved for the model(s) it may need to support for the application. The baseline use case is a model that can be completely served by one computer node. This is not possible on commodity infrastructure as tonnage of large models require partitioning using one of several strategies across multiple nodes in a compute mesh. In these cases, the “profiler” may need to engage a complex set of rules, heuristics, and decision tree involving different kinds of model partitioning possibilities as well as different compute mesh alternatives that could satisfy those strategies. When performance related parameters are supplied by the application such as token rate, that helps steer the profiler to mesh partition strategies that optimize overall inference latency.

These rules/heuristics/decisions could be baked into the “profiler”, but could also be a role served by the “resource manager.” In this way, the “resource manager” does more than just provide resource tracking and resource reservation capabilities.

3.3.2.4. Compose / Decompose

This stage involves:

- Model loading from repositories
- Creation and lowering of compute graphs
- Partitioning across the compute mesh

The system supports dynamic re-decomposition for fault tolerance and resource optimization.

At some point, the application has been profiled, the exact model has been identified, the model partitioning strategy has been selected, and the resource manager has reserved the mesh compute resources for this strategy. The polymorphic architecture for this application now moves to the “compose” stage where the model is loaded from a repository. If a model partitioning strategy was selected during the profile stage, the next stage is “decompose” in which the initial run-time high-level compute graph is created, the compute graph is lowered to an intermediate representation for the targeted hardware accelerator, and finally it’s partitioned across the components of the compute mesh reserved during by the “profiler.”. In this way, the “compose” is responsible for model retrieval, kick-starting the run-time,, while the “decomposer” is responsible for model partition hydration into the mesh compute memory(s).

Now that the system has loaded the model, the application can continue past model load into model inference. Model input data can now be streamed from the application to the loaded model’s associated compute mesh, which runs the model on data and returns the inference result back to the application. In the case of LLMs, input data is a prompt (set of tokens) and the output is the prompt completion (also a set of tokens up to the context window size). In the case where the batch is greater than one, the input and outputs have an extra dimension for the batch size.

In a highly dynamic and fault tolerant environment, esp. in one where compute nodes may go down due to hardware issues or connectivity issues, a running application might need to be “re-decomposed” into a different available mesh compute topology. This dynamism requires the following:

- A “resource manager” constantly monitors the health of available resources

- A “decomposer” that has the ability to tap into the “profiers” rules/heuristics/decision tree to quickly develop a new partition strategy and reserve an available compute mesh for the new partition

3.3.2.5. Free

When the application is done, the resources are returned back to the pool of available resources. The “resource manager” is involved in helping the “profiler” allocate and reserve resources, and should be involved in freeing up those resources.

4. High Level Architecture Definition

This chapter defines the polymorphic architecture's major functions and basic building blocks such that the polymorphic architecture can be constructed hierarchically and expanded via recurrent scaling-up and scaling-out. To facilitate the architecture simulation, these building blocks are described as data objects, and their state control / management as actions (APIs, e.g. compose, transform). These definitions are similar to an object-oriented programming model and made easy for simulation (e.g., using the simulation framework ASTRA-Sim[7]).

In this specification, we adopt the JSON-style declarations to describe all architectural resources, including compute, memory, and interconnect. Similarly, we define the state and management APIs using the named parameter convention such that APIs can be easily augmented without compatibility concerns.

4.1. Definitions of compute capabilities

Here, we define the basic computing elements which are treated as the last hierarchy of decomposition. That is, the basic computing elements comprise the most fine-grained instruction set (ISA) to establish the fundamental capability of the computing system. These computing elements can then be pooled together and re-composed into a more powerful logical Peip

4.1.1. Compute operators and interconnect types

Instead of a concrete list of instructions and data types supported by the architecture, here, we use a few coarse-grained operation categories and basic data type templates to define the capacity of Polymorphism Architecture through abstraction. The abstraction model avoids too many ISA details in the specification which pollute the mainstream of ideas, and it also makes the architecture extension and simulation relatively easy (Appendix E.2 has more discussion).

```
DATATYPE        := INT<x> (x|1-32) | FP<x> (x|4-64) |
                     BF<x> (x|8-16) | TF<x> (x|16-32) | LOG BASED
OPERATORS       := ARITHMETIC | GEOMETRIC | TRIGONOMETRIC |
                     CONV | MATMUL | GEMM | NORM | POOL |
                     TRANSFORMER | SUBGRAPH
```

Then we define the topologies and the parallelism mechanisms for how the basic computing elements are interconnected to form a new logical processor after re-composition. Note that

these topology and parallelism definitions are not directly related to the AI application's optimization strategies for various runtime parallelisms. Rather, they are used to describe how the compute elements communicate and collaborate with each other.

```
TOPOLOGY         := RING | ALL2ALL | 2DMESH | 3DMESH | CUBE | SWITCH
PARALLELISM      := PIPELINE | DATAFLOW | DSA | TENSOR| SEQUENCE

COLLECTIVE COMMUNICATION PRIMITIVES
                 := BROADCAST | SCATTER | GATHER | ALL-GATHER |
                    ALL-TO-ALL | REDUCE | ALL-REDUCE
```

With the definitions above, these logical processors can be recurrently recomposed at higher hierarchies, leading to the fractal computing architecture.

4.1.1.1. Compute resource description and APIs

The computing element can be conceptually treated as a chiplet: the basic compute unit in this hierarchical architecture. Its compute capability is fully determined by its own microarchitecture, the amount of physical resources, and intra-chiplet connectivity. Thus, we expect that the chiplet can only be partially allocated (a fraction of computation power) without restructuring its internal microarchitecture and such.

- Area: in mm^2
- Composability: [yes|no]
- Power consumption: {mean, static / dynamic}
    - Used for resource allocation strategies of perf per watts
- ISA: operators (list), average efficiency (mW/op)
- Supported data types: int32/int16/int8/int4/int2/int1, float point (64, 32, 16, 8, 4), other variants [11[
- Pooled compute:
    - Capacity: TOPs
    - Allocation granularity: integer percentiles (or ½, ¼, ⅓, ⅛, etc.)
    - Latency: compute and input/output in ns.
- Pooled memory:
    - Capacity: MB
    - Bandwidth: MB/s
    - Latency: in ns
    - Allocation granularity: integer percentiles
- Interconnect: [none] (i.e. ignore chiplet's internal connectivity)
- Ports: for external connectivity

    - Total bandwidth: MB/s
    - Bidirectional: [yes|no]
    - Granularity: integer percentiles (or ½, ¼, ⅓, ⅛, etc.)
- Parallelism: [none] (i.e. ignore chiplet's internal microarchitecture)?
- Topology: [none]

States and resource management APIs:

- allocate/free - call into this API from compute, or memory instances
- query - report the resource status (return capacity, available, used)
    - Resource availability, due to failure or fault, is reported to the upper level resource manager. Unavailability of resources may trigger dynamic re-configuration to bypass the failure and form a new compute topology.
- compose - assemble the resources based on the application characterization
- transform - architecture hardware optimization
- run - execute a compute task
- state := execution states (tops, compute, memory)

4.1.1.2. An example of compute resource allocation

As an example, we assume a NVDLA-like NPU with 32 TOPs of compute power. The complete description of the resource is as following:

```
ComputeElement {
    "Area"  : "20",
    "Power" : {15, 0},          // watts
    "Composability" : "No",
    "Interconnect"  : "None",
    "Topology"    : "None",
    "Parallelism" : "None",
    "Compute"    : "32",        // TOPs
    "Granularity": {1/2, 1/4, 1/8, 1},
    "ISA" : {ARITHMETIC, GEOMETRIC, CONV, MATMUL},
    "DataType" : {INT<16>, INT<8>, FP<16>, FP<8>},
    "Latency"  : "2",           // ns
    "Memory"   : "64",          // GByte
    "Mem-bandwidth" : "800",    // GB/s
    "Mem-latency"   : "5",      // ns
    "Mem-granularity" : {1/2, 1/4, 1/8, 1},
    "Ports" : "8",
```

```
    "Port-bandwidth"    : "64",    // GB/s
    "Port-granularity" : {1/2, 1/4, 1/8, 1}
};
```

The description above summarizes the high-level capability for the resource and how it can be partitioned (but no longer decomposable) with specified granularities.

Similarly, we show how to use the resource management API to allocate ⅛ of total resources:

```
ComputeElement.allocate (
    "compute-granularity"="1/8",
    "memory-granularity"="1/8",
    "port-granularity"="1/8",
    "instances"=8
);
```

Through this API (with named arguments), the resources (compute, memory, and connectivity) are equally divided into 8 instances, with 1⁄8 capacity per instance. The total allocated resource topologically forms 8 new logical processors as:

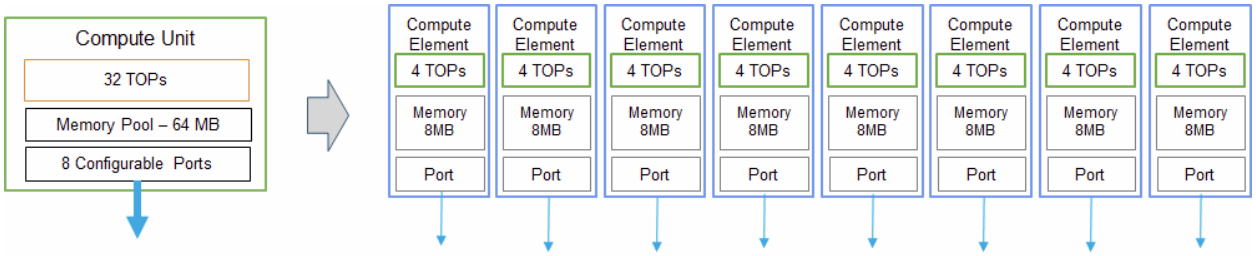


Figure 4. Resource allocation from the basic compute elements

This concept is similar to Nvidia's multi-instance-GPU (MIG) as we partition the resources into smaller instances to better match the application's runtime behavior.

4.2. Scale-up of compute units

Scaling up of compute units can be depicted as a set of computational resources connected on an efficient link to optimize the compute-communication paradigm. Examples may include

- CPU-GPU packaged element
- Computational elements in a single enclosure on a dedicated high-speed fabric
- Enclosures connected through an optimized topology to form a "pod"

Scaled-up compute units are composed from the base units (i.e., chiplets) using the description of a list of compute resources and a designated interconnect topology.

```
Compute capacity = hosting-compute-unit
                   + list [ worker-compute-units]
Configuration    = topology + interconnect-fabric
                   + list [ interconnect description]
Memory pool      = hierarchy | tiering
                   + list [ memory descriptor ]
```

Here, the hosting-compute-unit acts as a logical resource manager and compute task orchestrator. Worker-compute-units receive the task (or partitioned sub-tasks) distributed by the hosting-compute-unit and execute it. Worker-compute-units collaborate with each other through the interconnect fabric established by the topology configuration. Similarly, a pool of memory resources as provisioned using the memory descriptors are ready to be consumed, potentially across various memory tierings.

4.2.1. Scale-up description and APIs

States and managements APIs:

- allocate/free - resources (compute, memory, interconnect, etc.)
- query - capacity, available, used
- compose - assemble the architecture resources
- transform - architecture hardware optimization
    - Include dynamic tuning based on runtime profile (or learned strategy)
- run - execute a compute task
- state := (TOPs, compute, memory, interconnect, model values)
    - Track exception handling, error recovery, etc.
- Policies
    - Tenancy - spatial / time sharing of resources, data / fault isolation
    - SLA - to govern tuning
    - Power and cost - to govern resource allocation

A set of computing resources may be allocated from basic compute elements, within various compute nodes or even remote clusters. Then the total compute power of the entire designated pool of resources can be scaled up through composition into a virtually new compute accelerator.

4.2.2. An example of compute scale-up with composition

Using the example above, suppose the hosting computing unit is allocated with an interconnect fabric of 16 ports which each has a 64 GB/s bandwidth:

```
InterconnectFabric : fabric {
    "Ports" : "16",
    "Bandwidth" : "64",       // GS/s
    "Switch-latency" : "100", // ns
    "Switch-power" : "8",     // pJ/bit
    "Topology" : "SWITCH",    //
};

InterconnectLink : link {
    "Bandwidth" : "64",       // GB/s
    "Latency" : "100",        // ns
    "Bidirectional" : "Yes",  //
};

ComputeElement : host-compute {
    "Compute" : "16",
    "ISA" : {ARITHMETIC, GEOMETRIC, CONV, MATMUL, NORM, GEMM},
    "DataType" : {INT<16>, INT<8>, FP<16>, FP<8>},
    "Latency" : "2",
    "Memory" : "32",
    "Mem-bandwidth" : "800",
    "Mem-latency" : "5",
};
```

The resource pool can then be re-composed into the supported topology and be transformed into a more efficient logic accelerator for execution:

```
PooledComputing.compose (
    "Capacity" = {host-compute, ComputeElement.alloc(...)},
    "Configuration" = {fabric, link[8]}
    "Memory" = {MemoryPool.alloc(...)}
);
PooledComputing.transform( task.description() );
PooledComputing.run(task);
```

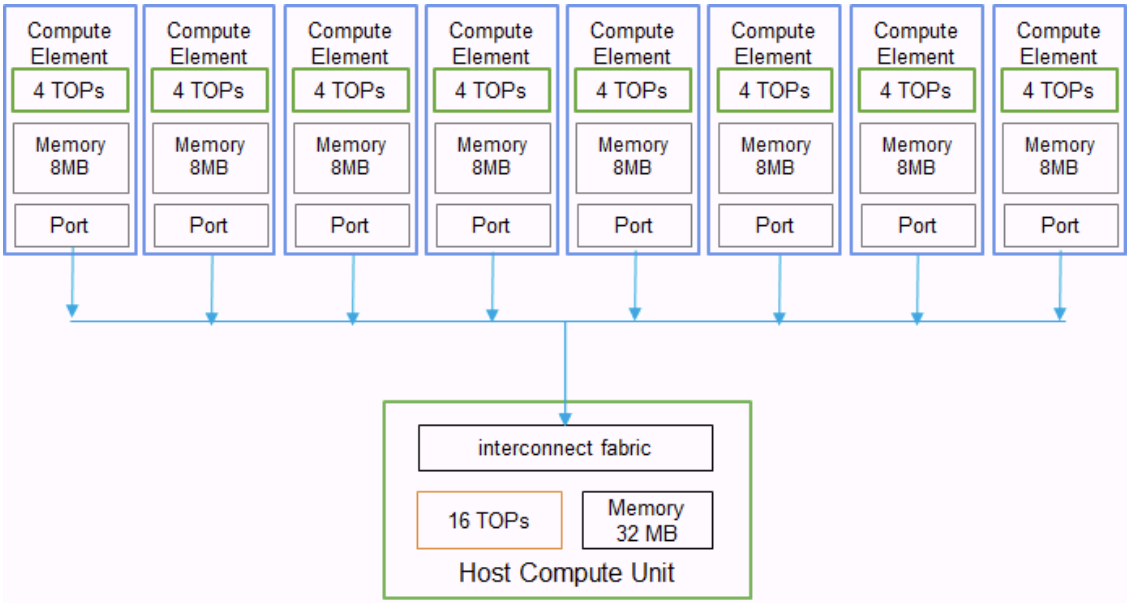


Figure 5. The allocated resources are scaled up to a new logical processor.

Figure 5 shows the architecture of the new logical accelerator. Since the interconnect fabric supports a switch-based topology, a number of collective communication algorithms may be applied to optimize the data movement among computing elements. Furthermore, the hosting computing unit may be optionally treated as a switch with the in-network computing capability. Thus, the SHARP-like reduction [8] can also be enabled to reduce data transportation latency even further.

4.3. Scale-out of compute for larger systems

To train larger models, there often is a need to interconnect the scale-up compute units and coordinate their execution to produce the end result. Scaling out of computing usually follows these steps:

- The overall problem space of the application is partitioned and optimized to map into a layout of computational elements
- The scale-up units are composed from the basic resources
- The scale-up units can then be further scaled out according to the interconnect configuration, forming a new type of compute resource just like a dynamically managed compute system.

```
Compute capacity = hosting-compute-unit
                   + list [ compute / scaled-up units]
Configuration    = topology + interconnect-fabric
                   + list [ interconnect description]
```

```
Memory pool       = hierarchy | tiering
                    + list [ memory descriptor ]
```

Hosting-compute-unit, similar to the Central Orchestrating Unit concept in some HPC edge cases, is needed here to schedule compute tasks and data transportation, pipelining the execution as well as avoiding network congestion.

States and managements APIs:

- allocate/free - compute, memory, interconnect, scaled-up resources
- query - capacity, available, used
- compose - assemble the architecture resources
- transform - architecture hardware optimization
    - Include dynamic tuning (via runtime profile or learned auto-tune)
- run - execute a computing task across distributed resources
- state := (tops, compute, memory, interconnect, model values)
    - Used for exception handling, error recovery, etc.
- Policies
    - Tenancy - spatial/time sharing of resources, data/fault isolation
    - SLA to govern tuning
    - Power and cost for resource allocation

To make scaling-out efficient, fast switching interconnects, such as the ULL Optical Switch (<1ns non-MEM) [15], may provide the flexibility for the system to optimize the workload and parallelize various sub-tasks among all resources. Similarly, composition and quick re-configuration of interconnects can help create polymorphism as the system is prepared for each new workload. This could even help disaggregate memory by using CXL over Ethernet. For scaling-out, the data distribution plane can be high bandwidth and the control plane can be low latency Ethernet.

4.3.1. Connecting Scale-up and Scale-out to build larger systems

A scale-up system can be viewed as a logical processor. As such, the system formed by interconnected scale-up logical processors can also be treated virtually the same as an individually scale-up unit itself. As scale-out goes on, fractal computing hierarchies are materialized.

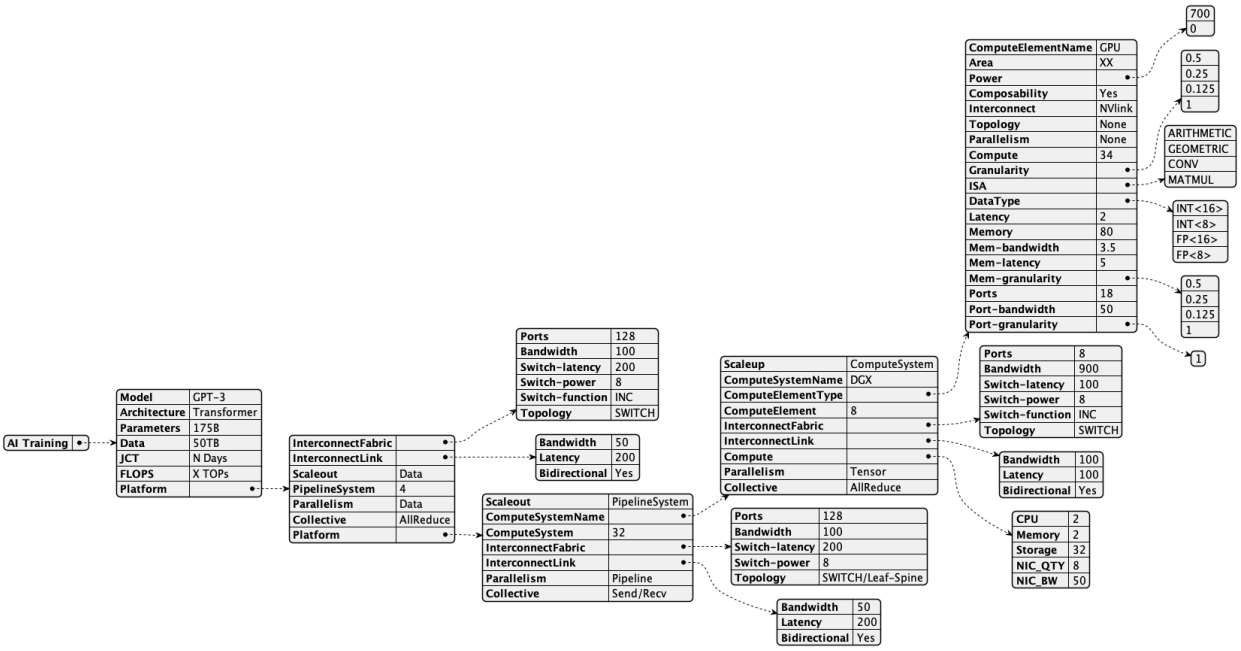


Figure 6. The JSON description of compute resources for training a GPT-3 model

For example, to pretrain a GPT-3 model, described as 175 billion parameters on 300 million tokens, the total FLOPs required can be calculated using transformer math. Using Figure 6 for FLOPs a user can determine how many GPUs are required to complete the pretraining in the required time. In the documented case this was 1024 A100 GPUs for 35 days. Composing the ecosystem to train this model is based on the ecosystem for the A100 from Nvidia. Nvidia offers the DGX based system that connects eight GPUs in one enclosure connected on a high speed, high bandwidth/low latency, NVlink/NVswitch interconnect. Eight GPUs is the baseline for the highest compute communication demand, tensor parallelism. However eight GPUs, each with 80GB of memory, cannot process the entire model due to memory size and time required for computation. The model is further disaggregated across DGX systems using pipeline parallelism, where the model is separated by layers in stages. With this 32 DGX systems are connected as a pipeline to process the model, forward/backward pass calculations, for a total of 256 GPUs in a pipeline. As the original compute-time calculation required 1024 GPUs four of the pipelines will be required to pretrain the model in the time required. These pipelines are composed using data parallelism, each pipeline calculates model data on different data batches using the same model information.

```
{
   "AI Training":{
       "Model" : "GPT-3",
       "Architecture" : "Transformer",
       "Parameters" : "175B",  //X TB of memory
```

```
        "Data" : "50TB",
        "JCT" : "N Days",
        "FLOPS" : "X TOPs",     //1024 GPUs
        "Platform": {
        "InterconnectFabric" :{
               "Ports" : "128",
               "Bandwidth" : "100",       // GB/s
               "Switch-latency" : "200",  // ns
               "Switch-power" : "8",      // pJ/bit
               "Switch-function" : "INC", // AllReduce
               "Topology" : "SWITCH"      // Rail-Optimized
        },
        "InterconnectLink" : {
            "Bandwidth" : "50",       // GB/s
            "Latency" : "200",        // ns
            "Bidirectional" : "Yes"
        },  //
        "Scaleout" : "Data",
        "PipelineSystem" : "4",
        "Parallelism" : "Data",
        "Collective" : "AllReduce",
        "Platform":{
            "Scaleout" : "PipelineSystem",
            "ComputeSystemName" : {
                "Scaleup" : "ComputeSystem",
                "ComputeSystemName" : "DGX",
                "ComputeElementType" : {
                    "ComputeElementName" : "GPU",
                    "Area" : "XX",
                    "Power" : [700, 0],         // watts
                    "Composability" : "Yes",
                    "Interconnect" : "NVlink",
                    "Topology" : "None",
                    "Parallelism" : "None",
                    "Compute" : "34",          // TFLOPs
                    "Granularity": [0.5, 0.25, 0.125, 1],
                    "ISA" : ["ARITHMETIC", "GEOMETRIC", "CONV",
"MATMUL"],
                    "DataType" : ["INT<16>", "INT<8>", "FP<16>",
"FP<8>"],
```

```
            "Latency" : "2",           // ns
            "Memory" : "80",           // GByte
            "Mem-bandwidth" : "3.5",   // TB/s
            "Mem-latency" : "5",       // ns
            "Mem-granularity" : [0.5, 0.25, 0.125, 1],
            "Ports" : "18",
            "Port-bandwidth" : "50",   // GB/s
            "Port-granularity" : [1]
        },
        "ComputeElement" : "8",
        "InterconnectFabric" :  {
            "Ports" : "8",
            "Bandwidth" : "900",       // GB/s
            "Switch-latency" : "100",  // ns
            "Switch-power" : "8",      // pJ/bit
            "Switch-function" : "INC", //AllReduce
            "Topology" : "SWITCH"      //
        },
        "InterconnectLink" : {
            "Bandwidth" : "100",       // GB/s
            "Latency" : "100",         // ns
            "Bidirectional" : "Yes"    //
        },
        "Compute" :  {
            "CPU" : "2",
            "Memory" : "2",    //TB
            "Storage" : "32",  //TB
            "NIC_QTY" : "8",
            "NIC_BW" : "50"
        },
        "Parallelism" : "Tensor",
        "Collective" : "AllReduce"
    },
    "ComputeSystem" : "32",
    "InterconnectFabric" : {
        "Ports" : "128",
        "Bandwidth" : "100",       // GB/s
        "Switch-latency" : "200",  // ns
        "Switch-power" : "8",      // pJ/bit
        "Topology" : "SWITCH/Leaf-Spine"    //
```

```
            },
            "InterconnectLink" :  {
                "Bandwidth" : "50",        // GB/s
                "Latency" : "200",         // ns
                "Bidirectional" : "Yes"    //
            },
            "Parallelism" : "Pipeline",
            "Collective" : "Send/Recv"
        }
        }
    }
}
```

4.4. Security Considerations

Machine learning based software workflow includes three main artifacts: Data, AI/ML Model, and Code. For instance, when training large language models for deployment, there are a number of actors performing actions on data over a span of time to produce the training results. This process can be subject to data manipulation or trojan injection to possibly influence the model output. However, to fully address these new dimensions of AI/ML data and model security issues is beyond the scope of this specification.

Instead, Polymorphic Architecture just focuses on how to solve data/fault isolation during resource sharing and re-composition such that failures in one application won't impact on other applications concurrently running on the same co-allocated resources. For additional information refer to Appendix C.

5. Memory Architecture Definition

This chapter defines the major functions and basic building blocks of the polymorphic architecture, enabling hierarchical construction and expansion via recurrent scaling-up and scaling-out. To facilitate architecture simulation, we describe these building blocks as data objects, with their state control and management as actions (APIs, e.g., compose, transform). These definitions align with object-oriented programming models and are suitable for simulation frameworks like ASTRA-Sim[7].

We adopt JSON-style declarations to describe all architectural resources (compute, memory, and interconnect) and define state and management APIs using named parameter conventions for easy augmentation without compatibility concerns.

In Polymorphic Architecture, a composable memory system is a unified abstraction of memory hierarchies for transparent memory tiering and provisioning. Memory composability allows scaling of memory capacity and bandwidth via disaggregated memory pools and/or direct access to memories residing in other remote compute nodes. Thus, we need a comprehensive description of memory resources such that the Polymorphic Architecture resource manager can allocate memory precisely for the current application according to its performance and other QoS metrics.

5.1. Definition of basic memory units

We define basic computing elements as the lowest hierarchy of decomposition, comprising the most fine-grained instruction set (ISA) to establish the fundamental capability of the computing system. These elements can be pooled and re-composed into more powerful logical processors.

Static description:

- Capacity: in MBytes
- Latencies: [read, write]. Access latencies in ns
- Bandwidth:
    - Native access (e.g. PCIe/DDRx): [MB/s]
    - Remote access (IPv6 or ROCe): [MB/s]
        - Remote bandwidth of 0 essentially makes it not shareable from non-local processors
- Properties: [transfer units, alignment, coherency, cache/buffer mode, power]
- Interface: protocol [compatible with DRAM/SRAM, HBM, CXL, NVMe, or other storage]
    - The need of caching or buffering would be determined from the characteristics of a particular workflow

- Computability: covers both near- and in- memory computing
    - Capacity: [flops]. Zero (0) means none of near/in-memory computing
    - Latency: [ns]. Zero (0) means in-memory computing
    - Bandwidth: [MB/s]. Infinity (∞) means in-memory computing

Management:

- isDivisible: [yes|no]. Can be partitioned or not.
- alloc/free: return the memory resources with the descriptions
- compose - compose the distributed memory allocations with a specified topology
- transform - optimize the memory usage according to the application's characteristics
- status - bookkeeping

5.2. Compute Operators and Interconnect Types

Instead of a concrete list of instructions and data types, coarse-grained operation categories and basic data type templates may be used to define the capacity of Polymorphic Architecture through abstraction:

```
DATATYPE     := INT<x> (x|1-32) | FP<x> (x|4-64) |
                BF<x> (x|8-16) | TF<x> (x|16-32) | LOG BASED
OPERATORS    := ARITHMETIC | GEOMETRIC | TRIGONOMETRIC |
                CONV | MATMUL | GEMM | NORM | POOL |
                TRANSFORMER | SUBGRAPH
TOPOLOGY     := RING | ALL2ALL | 2DMESH | 3DMESH | CUBE | SWITCH
PARALLELISM := PIPELINE | DATAFLOW | DSA | TENSOR | SEQUENCE
COLLECTIVE COMMUNICATION PRIMITIVES :=
                BROADCAST | SCATTER | GATHER | ALL-GATHER |
                ALL-TO-ALL | REDUCE | ALL-REDUCE
```

These definitions describe how compute elements communicate and collaborate, enabling recurrent recomposition at higher hierarchies to form fractal computing architecture.

5.3. Memory hierarchy and composability

In the process of building models there often is the case where the need for managing and storing data exceeds the capacity of the memory on the computational element. For this a memory system can be composed (from HBM, DRAM, NVMe, etc.) during the computation cycle (processing, training, tuning, pruning, inference). The logical memory system architecture is similar to the OCP CMS description here [29].

To efficiently utilize these layers in tiering the Memory Description will need to address

- Interconnect
- Locality
- Data Partition
- SLA

Simple memory (vs computational storage) is either permanent, long term or temporary storage for data. DRAM is fast but volatile storage and usually of limited capacity, so holds working data or is cache for slower data stores (like Flash). HBM is die-stacked DRAM, and is useful for storing larger amounts of data like temporary files, where the I/O for the HBM matches the speed at which data is generated or sunk, also as shared working store for a neural network being trained (that will later be committed to long term storage). HBM is a good choice for separating stages in a processing pipeline, when it isn't possible to stream directly. NVM like Flash is used for things like training data, which usually has fast read times, and slow write times. In training scenarios, data may be streamed directly from NVM rather than buffering it with DRAM, since the time to return to any individual item is long. So the attributes of a block of memory include:

- Size
- Access latency
- Throughput
- Fan-in/out (simultaneous users)

Many NVM drives automatically compress (and encrypt) data, and you want to avoid doing that twice. Encryption and compression add latency and consume power, so may be undesirable in some scenarios, in which case security concerns move to a higher level. The attributes of the data itself include:

- Lifetime
- Encryption (does it need to be secure)
- Compression

Figure 7 illustrates an example of a polymorphic memory system[4] with two types of primary memory, M1 and M2. M1 is located in the processing element and has better bandwidth and lower latency than the off-chip memory M2.The primary memory is logically partitioned by the memory address space and organized via the physical memory area. To take use of the cache and low latency of M1, a portion of M1 is employed as a DRAM cache of M2. It's also worth noting that in a multi-process and manycore environment, processes can compete for significant amounts of M1 memory in order to speed up the execution of their tasks. Allocating commonly used pages to M1 memory and resolving the contention issue between processes requires dedicated memory management to achieve optimal performance.

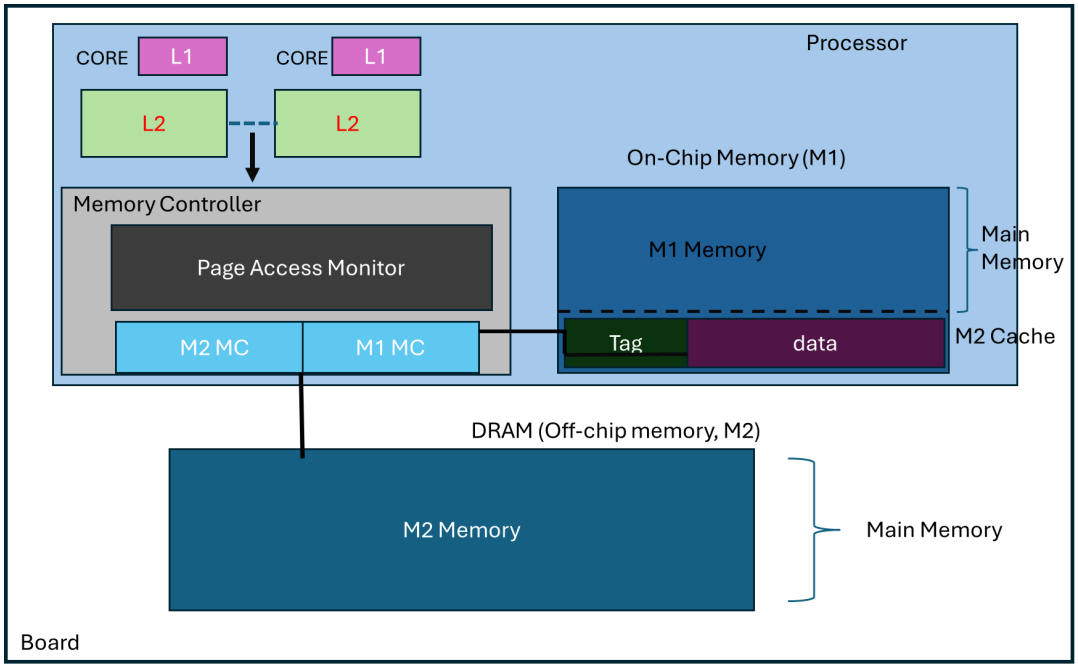


Figure 7 : Polymorphic Memory Architecture Example[4]

In a distributed/disaggregated computing environment, the processing element (or processor) may access not only to the local main memory (e.g., M1 and M2 in Figure 7), but also to the memory pool in the remote compute nodes. In the practical sense, it may not be beneficial to allocate the on-chip memory (i.e., M1) to a remote processor, but it is possible to make M2 available for sharing. In this case, making M2 partitionable and as a cache for remote memory could be a good optimization just as the optimization strategy in [4].

Using the memory definitions above, this memory architecture can be described as three basic memory units:

```
MemoryUnit-1.1 {
    "Capacity" : ?,
    "Latency" : {?, ?},
    "Bandwidth" : {?, 0},                    // not shareable
    "Properties" : {.., .., .., NONE, ..},   // regular memory
    "Interface" : SRAM,
    "Computability" : {0, .., ..},           // no in-memory compute
}

MemoryUnit-1.2 {
    "Capacity" : ?,
```

```
    "Latency" : {?, ?},
    "Bandwidth" : {?, 0},                    // not shareable
    "Properties" : {.., .., .., CACHE, ..}, // cache memory
    "Interface" : SRAM,
    "Computability" : {0, .., ..},           // no in-memory compute
}
```

With these two descriptions above, the first portion of M1 is for local use only (as a SRAM). The second portion of M2 is used as a cache. Both M1.1 and M1.2 have no in-memory computing capabilities.

```
MemoryUnit-2 {
    "Capacity" : ?,
    "Latency" : {?, ?},
    "Bandwidth" : {?, ?},                    // partitionable
    "Properties" : {.., .., .., NONE, ..},  // regular memory
    "Interface" : DRAM,
    "Computability" : {0, .., ..},           // no in-memory compute
}
```

M2 can be shared and accessed by both local and remote accelerators. Note that, the true caches (L1 and L2) are typically transparent to the applications. There is no need to describe them as memories.

5.3.1. Compute Resource Description and APIs

The computing element, conceptualized as a chiplet, is the basic compute unit in this hierarchical architecture. Its capabilities are determined by its microarchitecture, physical resources, and intra-chiplet connectivity. Key attributes include:

- Area (mm²)
- Composability [yes|no]
- Power consumption {mean, static/dynamic}
- ISA: operators (list), average efficiency (mW/op)
- Supported data types
- Pooled compute (capacity, allocation granularity, latency)
- Pooled memory (capacity, bandwidth, latency, allocation granularity)
- Interconnect and ports
- Parallelism and topology

Resource management APIs include:

- allocate/free
- query
- compose
- transform
- run
- state

5.3.2. Example: Compute Resource Allocation

Consider a NVDLA-like NPU with 32 TOPs of compute power:

```
ComputeElement {
    "Area"  : "20",
    "Power" : {15, 0},          // watts
    "Composability" : "No",
    "Interconnect"  : "None",
    "Topology"    : "None",
    "Parallelism" : "None",
    "Compute"    : "32",        // TOPs
    "Granularity": {1/2, 1/4, 1/8, 1},
    "ISA" : {ARITHMETIC, GEOMETRIC, CONV, MATMUL},
    "DataType" : {INT<16>, INT<8>, FP<16>, FP<8>},
    "Latency"  : "2",           // ns
    "Memory"   : "64",          // GByte
    "Mem-bandwidth" : "800",    // GB/s
    "Mem-latency"   : "5",      // ns
    "Mem-granularity" : {1/2, 1/4, 1/8, 1},
    "Ports" : "8",
    "Port-bandwidth"   : "64",  // GB/s
    "Port-granularity" : {1/2, 1/4, 1/8, 1}
};
```

To allocate 1/8 of total resources:

```
ComputeElement.allocate (
    "compute-granularity"="1/8",
    "memory-granularity"="1/8",
```

```
    "port-granularity"="1/8",
    "instances"=8
);
```

This API divides resources into 8 instances, forming 8 new logical processors (Figure 4).

5.4. Scale-up of Compute Units

Scaling up involves connecting computational resources on efficient links to optimize compute-communication paradigms. Scaled-up units are composed from base units (chiplets) using a list of compute resources and a designated interconnect topology:

```
Compute capacity = hosting-compute-unit + list [
worker-compute-units]
Configuration    = topology + interconnect-fabric +
                   list [ interconnect description]
Memory pool      = hierarchy | tiering + list [ memory descriptor ]
```

5.4.1. Scale-up Description and APIs

APIs for scaled-up units include allocate/free, query, compose, transform, run, and state management. Policies govern tenancy, SLA, power, and cost allocation.

5.4.2. Example: Compute Scale-up with Composition

[Includes the JSON examples for InterconnectFabric, InterconnectLink, and ComputeElement as provided]

The resource pool can be re-composed and transformed:

```
PooledComputing.compose (
    "Capacity" = {host-compute, ComputeElement.alloc(...)},
    "Configuration" = {fabric, link[8]}
    "Memory" = {MemoryPool.alloc(...)}
);
PooledComputing.transform( task.description() );
PooledComputing.run(task);
```

Figure 5 illustrates the resulting logical accelerator architecture.

5.4.3. Scale-out of Compute for Larger Systems

Scaling out interconnects scale-up units to train larger models. Steps include:

- Partitioning the problem space
- Composing scale-up units from basic resources
- Scaling out according to interconnect configuration

A hosting-compute-unit schedules tasks and manages data transportation.

5.4.4. Connecting Scale-up and Scale-out

Scale-up systems can be viewed as logical processors, forming fractal computing hierarchies when interconnected. Figure 6 provides a JSON description of compute resources for training a GPT-3 model, illustrating the composition of resources across multiple levels of scale.

5.5. Security Considerations

While machine learning workflows involve data, AI/ML model, and code artifacts, this specification focuses on data/fault isolation during resource sharing and re-composition. This ensures that failures in one application don't impact others running on co-allocated resources. For more details, refer to Appendix D.

5.6. Memory tiering

Polymorphic resource manager allocates individual units of memory from memory nodes across the memory hierarchies (local or remote) and binds them with compute resources to form a compute topology (using the compose API).

Polymorphic architecture runtime provides a unified transparent access to the allocated memory with normal memory access semantics. It leverages a similar mechanism of memory tiering as [30], which shows an over 70% GPU utilization improvement for the scenario of CXL memory offloading.

6. Software Architecture

Figure 8 captures a concrete implementation of the logic polymorphic architecture pipeline as depicted in Figure 2:

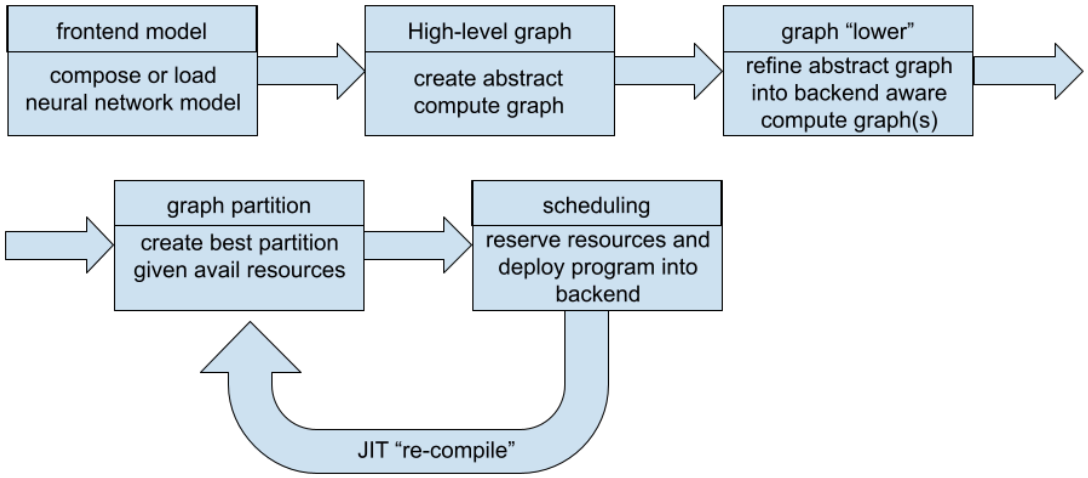


Figure 8. The high-level stages in Polymorphic Architecture pipeline

This pipeline makes these assumptions:

- Neural network models must be represented at runtime by a compute graph
- The compute graph, initially abstract and high-level, can be "lowered" to an optimal compute graph for each candidate hardware/compute resource available
- The system can then either:
    - Select the optimal compute graph/compute resource combination
    - Partition into sub-graphs, where each subgraph runs on different computing hardware
- Deploy the graph/subgraphs to the appropriate hardware resources
- As the model runs, the system detects conditions that could prompt a more optimal partition strategy and JIT-style "recompiles" the set of lowered compute graphs into a potentially better partition strategy

6.1. Basic primitives

6.1.1. Numeric data types

The data types supported by Polymorphic Architecture are mostly compatible with those defined in these sources [9, 10, 11].

However, this specification adopts basic data type templates as outlined in Section 3.1 to abstract Polymorphic Architecture away from any concrete implementation.

6.1.2. Aggregate types

Tensors

A tensor is a multi-dimensional array of elements, generally with the same numeric data type. Distinct types of tensors, in static or dynamic shapes, include:

- Model inputs and outputs (including batch size dimension)
- Model weights
- Model activations
- Model cache (e.g., KV cache for Transformers)

Quantized Tensors

Neural networks, especially large ones, are typically over-parameterized. They can be compressed (e.g., weight pruning, model distillation) or quantized (high precision/floating point weights/activations represented as compact integer types) with minimal loss to model accuracy.

To quantize a group of high precision, floating point weights to fixed point:

1. Select a group of numbers
2. Analyze to find a scaling factor
3. Apply scaling factor to all numbers so they fall within a fixed numeric range typically smaller than the original
4. Perform actual quantization (rounding to a "nearest" integer) within that transformed range

The group of numbers can be:

- The entirety of the neural network's weights
- Subsets across several layers
- A single layer of weights
- Even within a layer (e.g., channels in a convolution)

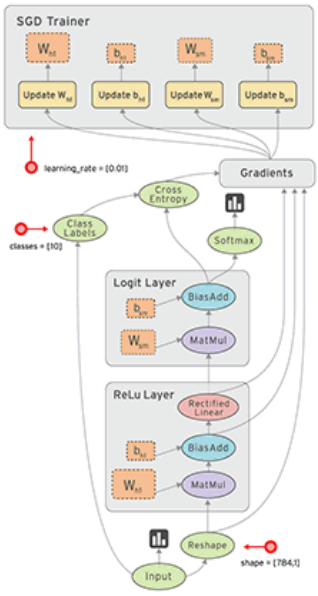
Common and emerging quantization formats should be efficiently represented by the basic data type templates, such as `INT<x>(x|1-8)` or `FP<x>(x|4-8)`, as detailed in Section 3.1.

6.2. Characterizing AI applications

6.2.1. Compute graph

High-level AI software frameworks (e.g., PyTorch, TensorFlow, ONNX) provide facilities for defining the topology of arbitrary neural networks using standard software primitives. These frameworks build a compute graph to capture the computational data flow of the model, either explicitly or implicitly. The compute graph is created statically at model load time or dynamically re-created at runtime (e.g., TensorFlow or PyTorch's "eager mode").

6.2.1.1. Abstract compute graph



High-level compute graphs start as simple data flow DAGs, where:

- Nodes represent mathematical operations on data
- Edges represent the data in the compute graph

Model inference traces the flow of data from model inputs, through various neural network layers, to model predictions. Pre-training adds an additional backward flow through the same graph, capturing "learning" in the form of model weight updates.

At a high level, a simple compute graph abstraction captures the parameter and data dependency for how AI models are trained, updated, and used in production.

The image at the left shows a simple 1-layer neural network as a compute graph in TensorFlow's TensorBoard. In practice, this high-level abstraction of a neural network model is not sufficient to optimize its operation in arbitrary compute backends, which may depend on many competing performance metrics such as:

- Latency
- Power consumption
- Memory budget
- Accelerator capabilities

- Resource availability

Therefore, contemporary large-scale AI compute pipelines have evolved beyond relying solely on the high-level compute graph abstraction. They now include additional intermediate representations that capture resource attributes and constraints of the "backend" target execution environment.

6.2.1.2. Graph lowering

The high-level compute graph resembles the layer-wise data flow of an arbitrary neural network and is often referred to as the "front-end" compute graph. It's a target-agnostic abstraction that gives way to downstream intermediate representations (IR) capturing the properties, capabilities, and limitations of the target execution environment for the model.

The next intermediate stage refines and aligns the compute graph with the capabilities of the target environment (also called the "back-end" compute graph). This process, essential in the Polymorphic AI specification, is referred to as graph lowering in AI pipelines.

Graph lowering for AI hardware accelerators

The examples of graph lowering include:

- Reassign graph node operators to optimized versions provided by the accelerator
- Fusing sub-graphs into smaller compute graphs or into one operator node
- Coercing numeric data into types more optimal for the hardware

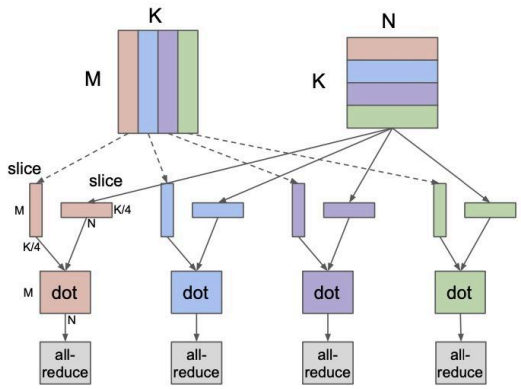
In general, hardware vendors develop custom backends or "kernels" to transform a high-level compute graph (the runtime representation of a model) into an optimal compute graph for their hardware. The lowered graph is typically the last level IR accelerated by hardware.

Graph partitioning via data sharding

The front-end/back-end IRs evolved during AI transformation when models could fit (or be coaxed to fit) into accelerator memory. However, the emergent properties of large models offered by "parameter scaling" necessitate extending compute graph abstraction to capture different model distribution and parallelization scenarios. This development is critical for training trillion-scale parameter models across thousands of accelerator nodes, such as TPU farms [19].

Early AI frameworks supported models exceeding monolithic memory by architecting model distribution capabilities, mostly orthogonal to the compute graph abstraction. Recent work (GShard [12] and GSPMD [13]) has demonstrated the idea of partitioning any data edge in the compute graph and assigning individual shards of data to compute nodes. A partitioning

strategy chooses the most strategic data edges, and the compiler infers and propagates actual partitioning through the remainder of the compute graph.



The example at left shows a partition strategy for two matrix data operands of matrix multiplication. The first matrix operand is partitioned 4 ways along the K dimension, each partition assigned to a different device. The second matrix is also partitioned along the K dimension accordingly.

Note that this manual labeling of the compute graph is necessary for distributing large models across compute nodes, but not sufficient to guarantee performance. Polymorphic Architecture leverages its offline AI profiling capability and real-time learned strategy to explore the partitioning space and achieve near-optimal performance.

6.2.1.3. Operators

Polymorphic Architecture supports operations similar to the ATEN IR [14] (one of two IRs for PyTorch 2.0 compute graph). Please see Appendix C for the recommended set of operators.

This specification uses a few coarse-grained operation categories to define the capacity of Polymorphic Architecture, as outlined in Section 3.1.

6.2.2. Model characterization

The compute graph (DAG) of AI models is refined and optimized to capture the most optimal runtime representation for the execution backend (AI accelerator). Depending on how the model is being used (inference, training, fine-tuning), Polymorphic Architecture may need to allocate resources differently for the task.

6.2.2.1. AI application states

KV Cache

Transformer-based large language models decode one token at a time (instead of ingesting the entire sequence at once) during inference. During decoding, the network typically maintains the state of previously decoded tokens in what's called a KV cache. This cache represents the "context" of the previously processed tokens in the sequence and enables Transformers to "auto-complete" the best responses for a prompt.

For large models, the cache is maintained to reduce response latency. This means a job scheduler needs to calculate the maximum cache size of a model (or sub-graph of the model if partitioned) to assign sufficient compute resources.

Optimizer State

AI training applications can consume significant accelerator memory due to various model training states (weights, gradients, and optimizer state). For example, when training a 1.5B parameter GPT-2 model, the Adam Optimizer [25] state (weights, first and second moment estimates) may consume up to 18GB, a significant portion of the accelerator memory (e.g., 32GB in an NVIDIA V100 GPU).

The size of application states needs to be factored in when allocating resources in Polymorphic Architecture.

6.2.2.2. Inference

The compute graph for inference is a feed-forward DAG. Data flows from input leaf nodes, which get transformed by internal operators in one direction and terminate at output nodes. Refer to Appendix B.1 for detailed inference flow.

Inference is often latency-bound. For example, in LLM inference, input is processed one element at a time, so there is insufficient computation to hide the latencies of memory and interconnects. Simultaneously, inference has a peak memory consumption directly proportional to the AI model size. Thus, Polymorphic Architecture should allocate resources with affinity of compute and memory to reduce latency.

Additionally, Polymorphic Architecture calls the transform API to adapt the resources and lower the graph - transforming it by replacing operator nodes with kernel-specific accelerated implementations, or fusing subgraphs into larger ones - to make the computation highly efficient.

6.2.2.3. Training

When training an AI model, the compute graph can be represented by a "joint graph" which combines two graphs:

- An "inference" feed-forward graph
- A sub-graph for backward propagation

The joint graph allows for:

- Tracking activation data computed during forward operation needed in the back-propagation step
- Back-propagating the training error so that weights can be updated

For LLM models, the amount of computation in training is proportional to the product of the model size and the training dataset size. For example, to train the GPT-4 model, the computation could amount to millions of exa-FLOPs (~10^25^). Similarly, the memory consumed by training may reach tens of terabytes (TBs).

Due to enormous computation and memory consumption, training requires substantial compute resources, potentially spanning tens of thousands of accelerators. Thus, interconnect bandwidth is another important factor to consider when allocating resources in Polymorphic Architecture. Simultaneously, AI application execution states may be partitioned and distributed among resources to avoid state redundancy or offloaded to various tiers of memory to enable more parallelism (e.g., ZeRO [23]).

6.2.2.4. Fine-Tuning

Fine-tuning has become a common operation for large language models. It's similar to training, except only a subset of the weights are updated when back-propagating the error. A slight variant is QLoRA [22], in which new parameters are injected into the network and only those are updated during the back-propagation stage.

6.3. Computing resource composition

Classical computers derive from Turing-machine type computing of single-threaded decision making. They usually involve fine-grained instruction sets to explore instruction level parallelism (ILP), such as out-of-order execution (OOO) and simultaneous multithreading (SMT). As Moore's Law for compute power scaling began to fade, computer architecture has trended towards domain-specific acceleration with more coarse-grained instruction sets, typically combined with single instruction multiple data (SIMD) or other vectorization technologies.

6.3.1. Fine-grained instruction sets

Fine-grained ISAs are often general-purpose, able to represent all types of operations from the application. Based on this abstraction, compilers (e.g., GCC and LLVM) can generate executable code for individual accelerator cores (e.g., shaders within a GPU). Popular examples of fine-grained ISAs include x86, ARM, and RISC-V.

Thus, the compute elements at the lower hierarchy of Polymorphic Architecture can support the most fundamental operations based on the fine-grained ISA. More complex operations, like many AI ops, can always be mimicked via decomposition into a sequence of fine-grained instructions. Decomposition will often require recompilation (on the fly) to achieve better performance.

6.3.2. Coarse-grained instruction sets

Coarse-grained instruction sets, often used by domain-specific accelerators (DSA), improve compute density by increasing multiple operations on a unit input of data. The architecture can include special computing elements with unique parallel mechanisms, data types, and domain-specific languages. By effectively utilizing native hardware computing capabilities, DSAs trade off architectural versatility to gain better performance and energy efficiency compared to fine-grained instruction set architectures. For example:

- NVIDIA Hopper GPU [18] and Google TPU [19] both include powerful tensor cores to speed up matrix multiplications, the most compute-heavy operations in AI applications.
- ISA-less processors (such as Ascenium [20] and NoISA Processor [21]) can be viewed conceptually as coarse-grained instruction set architectures. Resources of ISA-less processors are allocated and scheduled similarly to other conventional compute resources.

Additionally, non-von Neumann architectures (e.g., biological computing, memristor-based in-memory computing, and quantum computing as shown in Appendix F) have started to flourish. Since they are no longer based on sequential control flow execution, non-von Neumann architectures, as DSAs, can greatly improve performance by alleviating the memory wall effect.

6.3.3. Heterogeneous Systems

While domain-specific architectures have substantially increased computing power, they usually lack computational completeness. Therefore, heterogeneous computing is needed to combine the CPU's generality with the DSA's high efficiency. By allowing each DSA to maximize

performance in its own domain, heterogeneous computing can achieve the highest performance and best energy efficiency.

Heterogeneous computing systems typically include multiple kinds of DSAs, each with its own coarse-grained instruction set. Thus, it is important for each DSA to clearly advertise its compute capability, interconnect protocol, and bandwidth.

Polymorphic Architecture is a heterogeneous compute system composed of:

- Clusters of DSAs
- Interconnect resources (e.g., NVLink/NVSwitches)
- Memory hierarchies (local and/or remote)

These resources may be decomposed or re-composed using the strategies described in Sections 3.1 and 3.2 to enable more versatile compute capabilities and higher computer efficiency.

6.3.4. Computing topology auto-detection and transformation

As shown in Figure 9, in Polymorphic Architecture, common element types are grouped into islands. To manage work and configuration of the system, the elements need to be discovered and cataloged to an inventory of resources in the context of an island. Resources can be managed locally and mapped as capabilities into a service endpoint or API abstraction in the topology. Then, applications can be best executed for that locality or under a given set of service constraints.

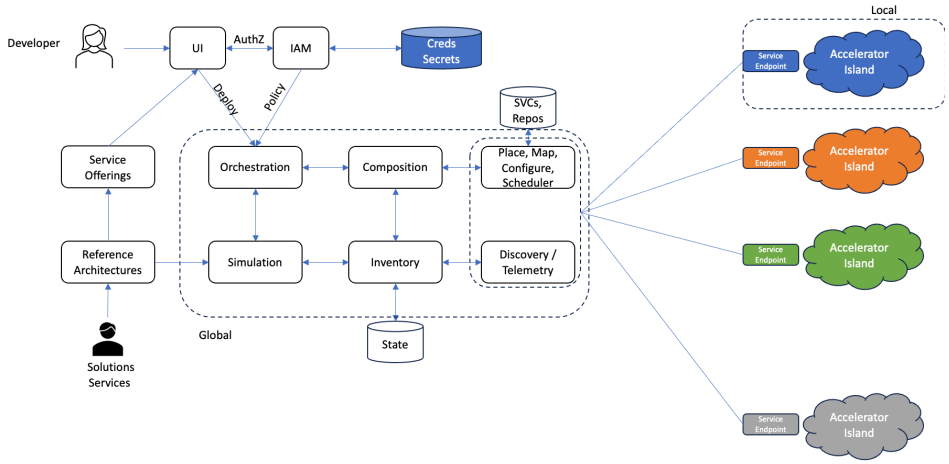


Figure 9. Polymorphic architecture high-level resource configuration

Figure 10 shows a detailed execution flow:

1. A user request, authenticated and mapped to a policy, is deployed to an orchestration engine.
2. The orchestration engine simulates (or learns) possible composition configurations based on the request, resources, capabilities, and available inventory.
3. Simulation results are returned, from which the orchestration engine determines the best possible paths.
4. Work is delivered to a composition engine.
5. The composition engine consumes inventory and schedules work through an API to the service endpoint for the accelerator island to perform the work.

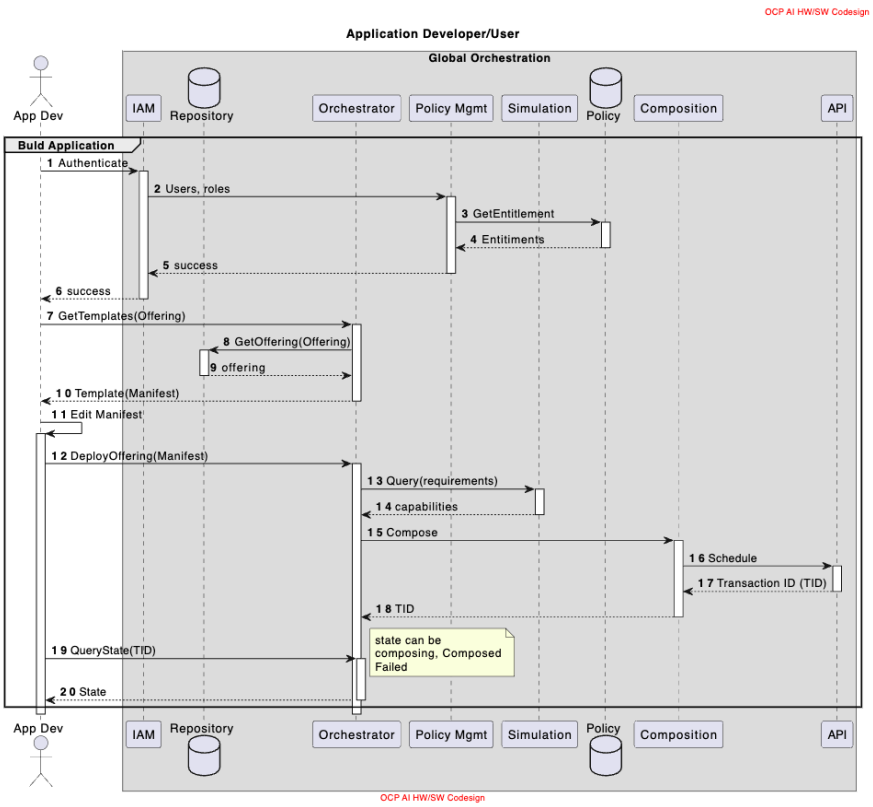
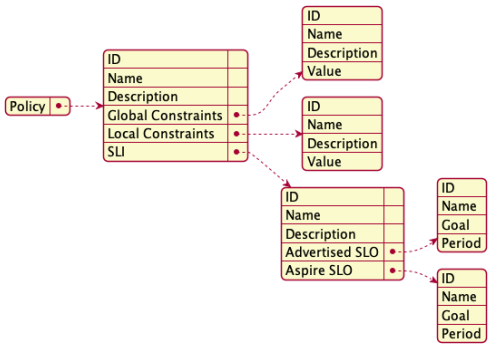


Figure 10.The application's detailed execution flow.

To facilitate resource and topology discovery, this architecture leverages the OpenFabric Management (Sunfish) [24] proposal for heterogeneous network management. Sunfish builds

on existing standards to enable discovering and managing fabrics for Ethernet, CXL, PCIe, IB, Slingshot, and is extensible for proprietary interconnects. All of these can be simulated, and topology possibilities can be discovered and configured.



Policy governs both the user request to compose resources and the required service level for executing the task. Global constraints apply to the user's ability to use resources and data, as well as the user request that can map to time to complete, cost, data access, etc.

Figure 11 provides detailed flow for resource discovery.

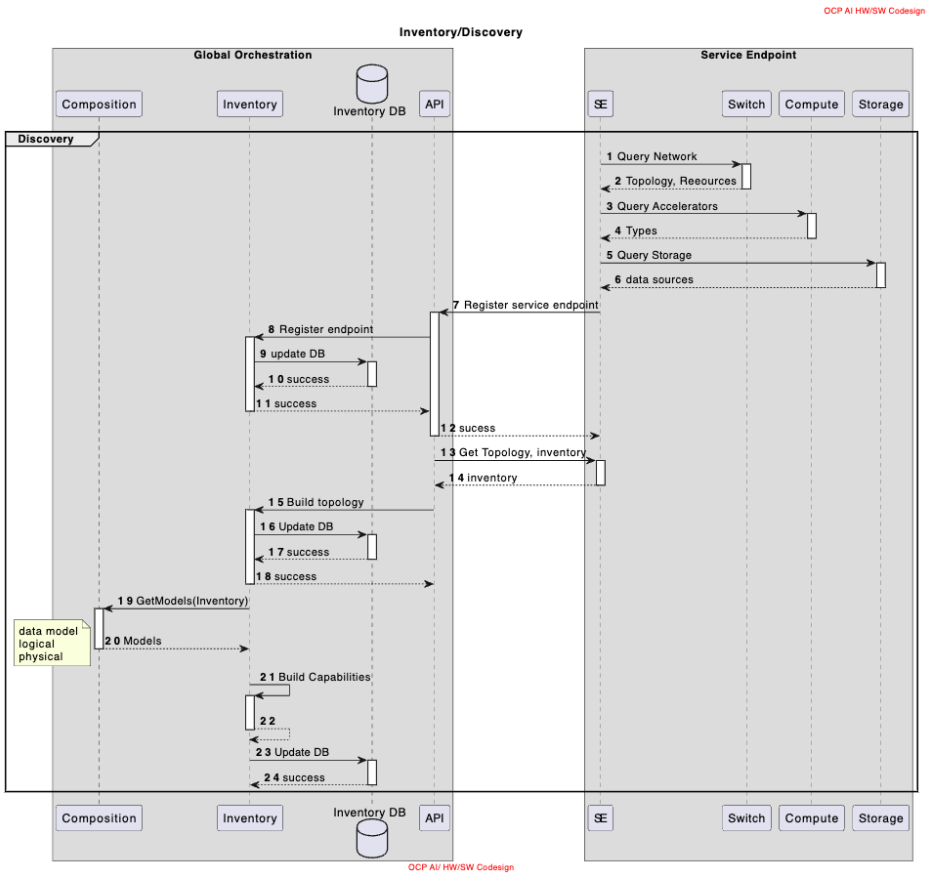


Figure 11.The resource discovery flow.

6.4. Computation partitioning and scheduling

The AI model at run-time is a compute graph intermediate representation (IR) refined for the optimal backend execution. For example, an initial high-level graph might not define the size of a batch (ie, it's a wild-card dimension,) but the final "lowered" graph run by an AI accelerator will need exact tensor dimensions. One might be able to view the final "lowered" compute graph as approaching an embedded program in which performance is highly predictable (no branching, well-defined tensor sizes, specifically parameterized kernel code references, etc.)

Wherever possible, Polymorphic Architecture should leverage the final "lowered" compute graph to make the best scheduling decisions for the workload based on the available resources. To accomplish this, it needs the following:

- The aforementioned "lowered" compute graph:
    - This means that the scheduler needs the ability to access the "lowered" compute graph for potentially different execution backends (different AI accelerators participating in the work pool).
- Constant monitoring of available resources and the ability to reserve those resources for assignment. Besides tracking if the resource is available or busy, a scheduler will need to track:
    - Type of compute resource (ie, type of GPU and how its attached to a host)
    - For partitioned workloads, how compute nodes are connected to each other and the type of communication primitives and its security and reliability
    - For multi-tenant resources, how much memory is available, how many processing cores/elements are available, and how the processing cores are shared (time-sharing or physically partitioned). Note that resource sharing is subject to the policy constraint of fault isolation.

With this level of information (the "lowered" compute graph) and visibility of the resource pool, the scheduler can make informed compute graph partitioning decisions and can confidently assign sub-graphs to the best resources (and utilize the resources most effectively.)

In this vein, there are several patterns of mode/compute graph partitioning worth highlighting here.

6.4.1. Deep Learning Parallelisms

In practice, both AI application model and data can be partitioned. A scheduler could be programmed with a set of (learned) heuristics or decision tree to hone in on the right partitioning strategy. Since it might be difficult to assess the cost of yet unseen work partitioning strategies,

the schedule could keep a running memory of actual cost observed so that it can learn to improve over time.

6.4.1.1. Data Parallelism

This simplest parallelism is to replicate the model/compute graph in multiple compute resources, but partition the input data among those resources. This strategy minimizes the communication overhead during processing as only the outputs need to be joined at the conclusion. Unfortunately it does require a large run-time memory footprint since the entire model is replicated (not a great solution for billion-scale/trillion-scale parameter models.)

For AI training, data parallelism also requires high interconnect bandwidth to reduce the latency for gradient reduction (all-reduce).

6.4.1.2. Pipeline Parallelism

This is the most straightforward model parallelism in which you partition groups of consecutive layers in the model, and allocate groups individually to different compute resources. Each group processes its batch in turn and feeds the results to the next group in the model sequence. The following graphic is a great depiction of pipeline parallelism:

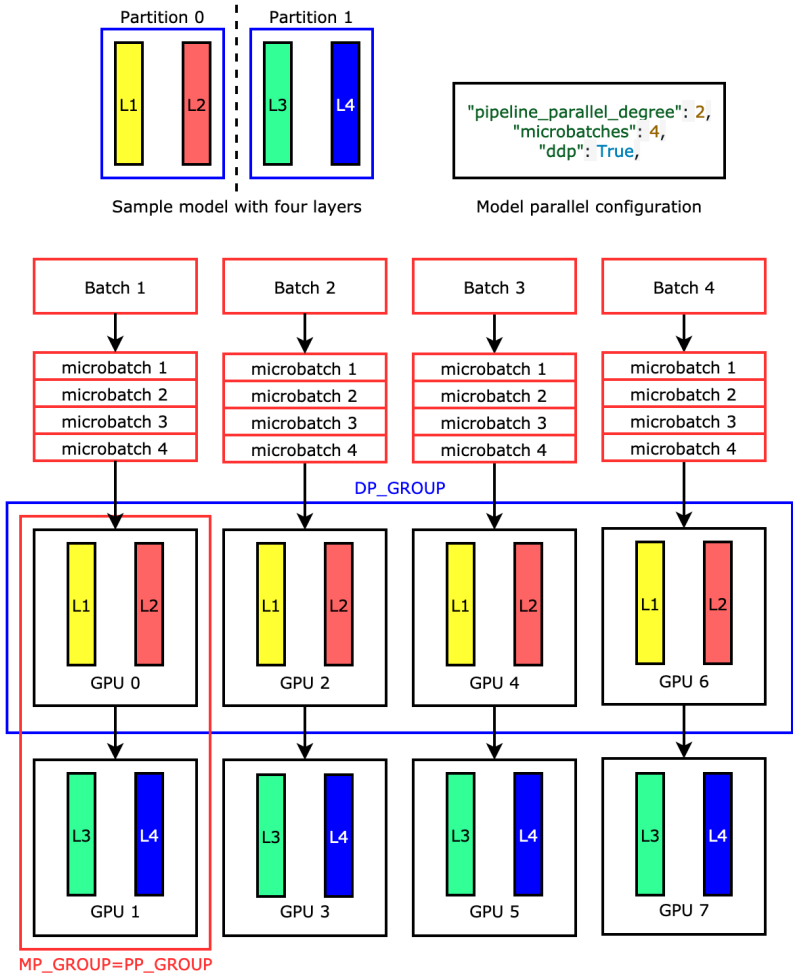


Figure 12.Model (pipeline) parallelism and data parallelism [26]

6.4.1.3. Tensor Parallelism

This is necessary for large models because 1-layer can be composed of billions of parameters. For example, one of the most common tensor parallelism use cases is to split the feed forward in the decoder block of large Transformers. Tensor parallelism can be the most complicated due to data synchronization requirements during the operation. For example, a large matrix multiplication contains multiple SUM/REDUCE operations and, depending on the partitioning strategy, the operands may live on different compute nodes. Figure 13 depicts this kind of tensor parallelism:

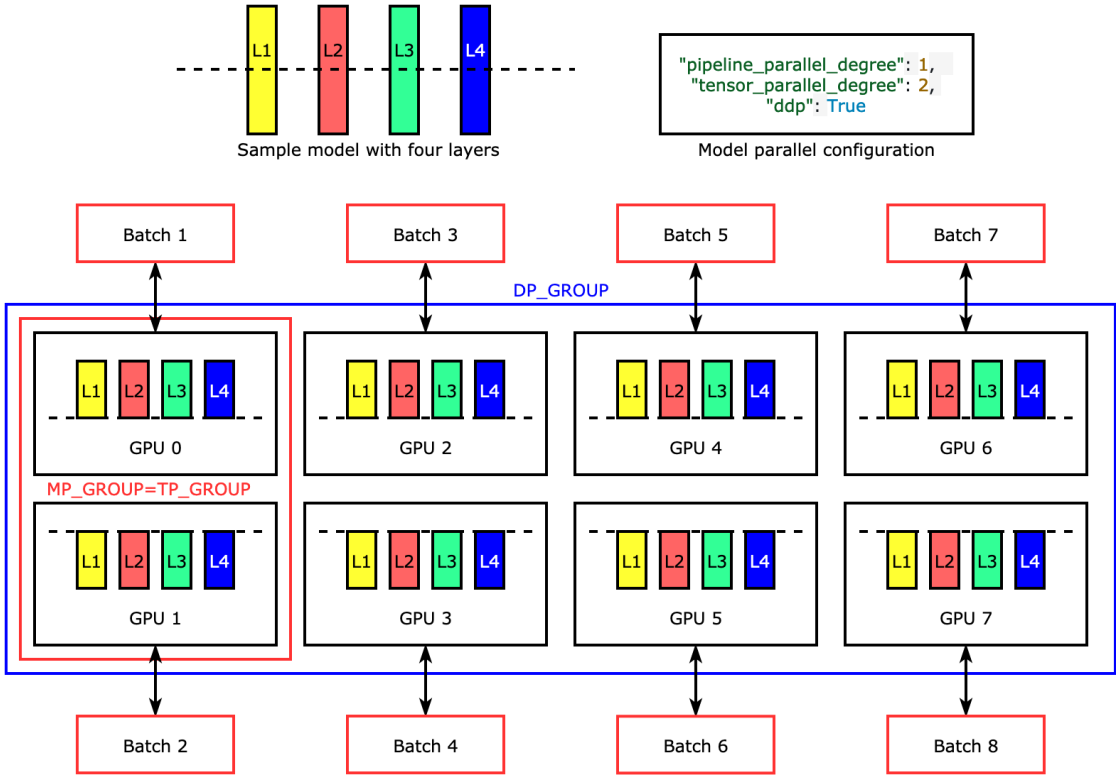


Figure 13. Model (tensor) parallelism [26]

6.4.2. Collective communication primitives

The above compute-communication mechanisms are enabled in the programming space through Domain Specific Languages (DSL). Each of the collectives used to create the parallelism described above can be incorporated in program to optimize the communication mechanism based on the graph computed for the application to be executed. One such collective communication library is NCCL, from Nvidia, utilized to communicating data between GPUs for computing results. There are derivative implementations of NCCL, tuned to specific data center implementations. Microsoft's Collective Communication Language (MSCCLang) [6] provides a unified system to customize communication algorithms and generate a high-performance implementation of collective communication. MSCCLang defines a domain-specific language (DSL) to describe various communication patterns. It also includes a runtime with NCCL-compatible APIs - inherit Nvidia NCCL's support of a diverse set of GPUs and interconnects.

Polymorphic Architecture will leverage the DSL approach, referencing the Unified Collective Communication (UCC) standards proposal as an example, to describe the fine-grained

optimization and communication patterns such that the optimal data movement can be achieved after resource composition.

7. Architecture APIs for Simulation

The simulation of Polymorphic Architecture involves a set of actions with prescribed intents against a set of resources. These resources can be:

- Single elements (compute, communication, memory, data)
- Composed sets of resources to perform more complex functions

In the ecosystem, resources are connected in a topology. Each resource is identified by a Globally Unique Identifier (GUID) and has a defined relationship within the topology. When resources are composed to perform a given function, that function can also be identified by a GUID, mapping to a set of resources in the context of a topology. This context can be represented as a subgraph of the data center topology, corresponding to the computational graph representing the action to be performed.

The hierarchical approach allows querying of:

1. Jobs, Actions, Functions
    - Resources assigned, topology
    - Intent, SLA, constraints
2. Resources composed
    - Capability
    - Capacity
    - Topology
3. Resources discovered
    - Capabilities, properties
    - Capacity
    - Topology

A sample resource hierarchy can be described as follows:

```
ResourceHierarchy {
  List of resources [
    compute capacity,
    memory capacity,
    interconnect capacity,
    supported ISA
  ];
  Resource topology descriptor {
    Topology via composition;
    List of interconnect properties;
```

```
  };
}
```

The interconnect topology can be composed hierarchically using network building blocks similar to those in ASTRA-Sim [7]. For example, a 32-GPU compute system might consist of four nodes, each with 8 GPUs interconnected as a ring. The four nodes are then connected by a switch to enable an all-to-all connection. The topology of this system can be composed as R(8)-SW(4).

PolyArch extends this topology composition notion to form even larger systems by compositing multiple sub-interconnect-topologies.

7.1. Preliminary high-level APIs for simulation

Here are preliminary high-level APIs for simulation:

```
// Properties includes computation graph elements, constraints,
// time to completion, cost, resources.
Properties * getProperties(App *);

// Pool is defined with resources GUID, compute resource
// descriptions, memory descriptions, interconnect descriptions,
// interconnect topology, locality.
Resources * allocResources(Properties *appProp, Pool *);

// App may contain intent in addition to resource requirements
Boolean isSupported(App *, Resources *);

// Include the resource binding among compute, memory, and
// interconnect with the specified topologies.
LogicalResources * composeResources(Resources *);

Boolean transformResources(Properties *, LogicalResources *);

// Program is composed of compiled functions in App.
Executable * compileProgram(Program *, ResourceDescription *);

Boolean executeProgram(Executable *, LogicalResources *);
void releaseResources(Resources *);
```

```
Device-level APIs:
// Devices
Resources *getAvailableResources(Device *);
Interconnect *allocateAvailableInterconnect(DeviceInterconnect *);
.. .. ..
```

7.2. Reference implementation of simulation

PolyArch potentially consists of many distributed pools of resources, including:

- Heterogeneous compute accelerator pools
- Memory/storage pools
- Interconnect pools

Therefore, PolyArch needs to abstract away resource heterogeneity with a set of definitions for operations and data types. The implementation of PolyArch can be based on either:

1. A very detailed simulation using the definitions from Appendix C
2. A high-level cycle-approximate simulation using the coarse-grained operator categories and data type templates from Chapter 3

The cycle-approximate simulation is feasible to achieve a very large-scale system with much reduced simulation time.

Appendix E shows a reference simulation framework that implements the key components of PolyArch. To support heterogeneous compute hardware, the low-level vendor-specific software components (or property descriptions) should be available so that the high-level PolyArch architecture abstraction can be constructed.

Accordingly, the PolyArch compiler (JIT compilation) should also be implemented to compile the AI model operators into executable code on the simulator. Alternatively, an analytic execution could be conducted to calculate the approximate execution time and resource consumption.

7.3. ASTRA-Sim plug-ins

PolyArch simulation can leverage the ASTRA-Sim infrastructure [7], primarily developed by Intel, Meta, and Georgia Tech. The ASTRA-Sim 2.0 framework provides the following capabilities, making it suitable for simulating the large-scale Polymorphic Architecture:

1. Multi-dimensional network with various topology representations at scale

2. Distributed memory fabric supporting both local and remote memory hierarchies
3. Arbitrary parallelization strategies, allowing Polymorphic Architecture to exercise different optimizations across its composable compute hierarchies

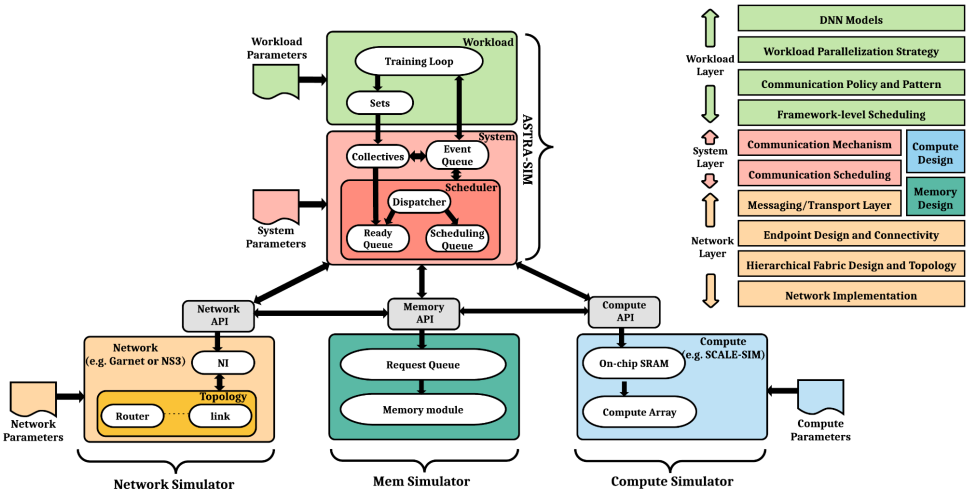


Figure 14. ASTRA-Sim high-level workflow [7]

The PolyArch simulator can be part of a joint effort between:

- OCP AI Co-Design workstream
- OCP community
- MLCommons Chara workgroups

A set of plug-in APIs should be designed to interface the PolyArch architecture with the ASTRA-Sim framework. The framework should also be extended to simulate the composability and transformability of PolyArch.

8. Conclusion

Polymorphic architecture represents a novel computing paradigm designed to address the rapidly evolving landscape of AI applications. This architecture is characterized by two fundamental properties:

1. Continuous transformability
2. Recursive Composability

These properties extend across vertical hierarchies of computing resources, spanning from chiplets to compute nodes, server systems, large-scale clusters, and beyond.

This specification proposal defines the high-level functions and execution flow of a Polymorphic Architecture for AI systems. It proposes microarchitecture definitions at the block level, describing how compute and interconnect resources can be decomposed and shared. This enables recursive resource composability and transformability. It recommends a Polymorphic Architecture programming model and associated software system, including a structure of APIs to enable cycle-approximate simulation. It discusses the potential of leveraging existing simulation frameworks to demonstrate the advantages of polymorphic architecture. And finally it calls for community action to standardize architectural simulation APIs, with the goal of establishing a robust polymorphic computing ecosystem.

The Polymorphic Architecture approach offers several significant benefits:

- **Flexibility**: It can adapt to the changing requirements of AI workloads more effectively than traditional fixed architectures.
- **Efficiency**: By allowing resources to be dynamically composed and transformed, it potentially offers better resource utilization and performance optimization.
- **Scalability**: The fractal nature of the architecture allows it to scale from small systems to massive, distributed computing environments.
- **Future-proofing**: Its adaptable nature makes it well-suited to accommodate emerging AI techniques and models as well as provides a vehicle for the realization of truly heterogeneous component utilization,

However, realizing this architecture presents several challenges:

Firstly, managing the dynamic composition and transformation of resources adds significant complexity to both hardware and software systems. Secondly, widespread adoption would require **substantial standardization efforts** across the industry. Lastly the flexibility of the

system could potentially introduce performance overheads in various scenarios that need to be carefully weighed and managed. These challenges are non trivial, however there are solid foundations and approaches to resolution proposed within this white paper.

Moving forward, the success of Polymorphic Architecture will depend on:

1. Further refinement and validation of the concepts presented in this specification
2. Development of robust simulation and prototyping tools
3. Collaboration among hardware vendors, software developers, and AI researchers
4. Real-world testing and benchmarking to demonstrate tangible benefits over existing architectures

In conclusion, while Polymorphic Architecture presents a promising approach to addressing the evolving needs of AI computing, significant work remains to fully realize its potential. This proposed approach is to serve as a foundation for future research, development, and standardization efforts in this exciting field.

9. Glossary

Listed in alphabetical order.

Term	Description
AI	Artificial Intelligence
API	Application Programming Interface
ASTRA-Sim	Advanced System for Training and Research Analysis Simulator
BFP	Brain Floating Point
BMC	Baseboard Management Controller
BNF	Backus-Naur Form
CMS	Composable Memory System
CPU	Central Processing Unit
CXL	Compute Express Link
DAG	Directed Acyclic Graph
DCN	Data Center Network
DDR	Double Data Rate
DECOMPOSITION	This refers to a process where resources are broken down to smaller chunks. For example, a large GPU can be partitioned into multiple instances, each of which can work as an independent (less powerful) GPU.
DISAGGREGATION	This refers to the state of computer resources which are physically separated, typically according to the resource categories. For example, a pool of CPUs can reside in one rack while a pool of memory (or storage) resides in a different rack.
DRAM	Dynamic Random Access Memory
DSA	Domain-Specific Architecture
DSL	Domain-Specific Language
FPGA	Field-Programmable Gate Array
GDPR	General Data Protection Regulation
GPU	Graphics Processing Unit

GUID	Globally Unique Identifier
HBM	High Bandwidth Memory
HPC	High-Performance Computing
ICI	Inter-Chip Interconnect
ILP	Instruction Level Parallelism
IR	Intermediate Representation
ISA	Instruction Set Architecture
ISLAND	Grouping of common element types in a given system or as a subsystem
JIT	Just-In-Time
KV	Key-Value
LLM	Large Language Model
MSCCL	Microsoft Collective Communication Library
NCCL	NVIDIA Collective Communications Library
NFS	Network File System
NVMe	Non-Volatile Memory Express
OCP	Open Compute Project
OOO	Out-of-Order
PAIA	Polymorphic AI Architecture
PCIe	Peripheral Component Interconnect Express
QC	Quantum Computing
QoS	Quality of Service
QLORA	Quantized Low-Rank Adaptation
RLHF	Reinforcement Learning from Human Feedback
RAG	Retrieval Augmented Generation
RAS	Reliability, Availability, Serviceability
RDMA	Remote Direct Memory Access
SAN	Storage Area Network
SHARP	Scalable Hierarchical Aggregation and Reduction Protocol
SIMD	Single Instruction, Multiple Data
SLA	Service Level Agreement

SMT	Simultaneous Multithreading
SSD	Solid State Drive
TEE	Trusted Execution Environment
TPU	Tensor Processing Unit
ULL	Ultra-Low Latency
VRAM	Video Random Access Memory
XLA	Accelerated Linear Algebra

10. References

1. OCP AI Co-Design whitepaper, “Polymorphic architecture for future AI applications”, 2023
2. Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. Multi-tenant GPU clusters for deep learning workloads: Analysis and implications. *Technical report, Microsoft Research*, 2018.
3. Paul Barham, Aakankasha Chowdhery, Jeff Dean, Sanjay Ghemawat, et al, “Pathways: Asynchronous Distributed Dataflow for ML”, Proceedings of the 5th MLSys Conference, 2022
4. Lim, SeungHo, Seok, Hyunchul and Park, Kyu Ho, “Polymorphic Memory: A Hybrid Approach for Utilizating On-Chip Memory in Manycore Systems”, Electronics 9(12), 2020. DOI: 10.3390/electronics9122061
5. Jeff Bryson, “Polymorphic System Architecture”,
6. Meghan Cown, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, Yifan Xiong, “MSCCLang: Microsoft Collective Communication Language”, the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-23), 2023
7. William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, Tushar Krishna, “ASTRA-Sim 2.0: modeling hierarchical networks and disaggregated systems for large-model training at scale”, ISPASS 2023
8. Nvidia, “Nvidia Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) Rev 3.0.0”, https://docs.nvidia.com/networking/display/sharpv300
9. Common IEEE floating point format, https://en.wikipedia.org/wiki/IEEE_754
10. PyTorch 2.0 tensor data types, https://pytorch.org/docs/stable/tensors.html

11. OCP MX-compliant data types, https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf
12. D.Lepikhin, H.Lee, Y.Xu, D.Chen, et al, “GShard: scaling giant models with conditional computation and automatic sharding”, https://arxiv.org/abs/2006.16668
13. Y.Xu, H.Lee, D.Chen, B.Hechman, et al, “GSPMD: general and scalable parallelization for ML computation graphics”, https://arxiv.org/abs/2105.04663
14. PyTorch ATEN IR, https://pytorch.org/docs/main/torch.compiler_ir.html
15. Agitron ultra low latency optical switch, https://agiltron.com/product/low-latency-optical-router/
16. H.Touvron, L.Martin, K.Stone, P.Albert, et al, “Llama 2: open foundation and fine-tuned chat models”, https://arxiv.org/abs/2307.09288
17. Hugging face: the AI community, https://huggingface.co/
18. Nvidia Hopper Architecture, https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture/
19. Google Cloud Tensor Processing Units (TPU), https://cloud.google.com/tpu
20. Ascenium Aptos, “A new class of general purpose processor”, https://ascenium.com/
21. NoISA Processor, https://hotwright.com/
22. T.Dettmers, A.Pagnoni, A.Holtzman, L.Zettlemoyer, “QLoRA: efficient finetuning of quantized LLMs”, https://arxiv.org/abs/2305.14314
23. Microsoft Zero Redundancy Optimizer (ZeRO), https://www.deepspeed.ai/tutorials/zero/
24. OpenFabric Management framework, https://www.openfabrics.org/openfabrics-management-framework/
25. D.Kingma, J.Ba, “Adam: a method for stochastic optimization”, https://arxiv.org/abs/1412.6980
26. Amazon AWS, “Introduction to Model Parallelism”, https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-intro.html
27. ACE Center for Evolvable Computing, https://acecenter.grainger.illinois.edu/
28. Unified Acceleration Foundation (UXL), https://uxlfoundation.org./
29. OCP CMS Logical System Architecture white paper, https://docs.google.com/document/d/18kiTJEEGfFr3vAGa5xJuja9_HKfrxg6iXL5F2G8Ug90/
30. MemVerge and Micron boost NVidia GPU utilization with CXL memory, GTC 2024, https://memverge.com/gtc2024/

31. OCP ODSA chip data exchange markup language, https://github.com/opencomputeproject/ODSA-CDXML

11. License

OCP encourages participants to share their proposals, specifications and designs with the community. This is to promote openness and encourage continuous and open feedback. It is important to remember that by providing feedback for any such documents, whether in written or verbal form, that the contributor or the contributor's organization grants OCP and its members irrevocable right to use this feedback for any purpose without any further obligation.

It is acknowledged that any such documentation and any ancillary materials that are provided to OCP in connection with this document, including without limitation any white papers, articles, photographs, studies, diagrams, contact information (together, “Materials”) are made available under the Creative Commons Attribution-ShareAlike 4.0 International License found here: https://creativecommons.org/licenses/by-sa/4.0/, or any later version, and without limiting the foregoing, OCP may make the Materials available under such terms.

As a contributor to this document, all members represent that they have the authority to grant the rights and licenses herein. They further represent and warrant that the Materials do not and will not violate the copyrights or misappropriate the trade secret rights of any third party, including without limitation rights in intellectual property. The contributor(s) also represent that, to the extent the Materials include materials protected by copyright or trade secret rights that are owned or created by any third-party, they have obtained permission for its use consistent with the foregoing. They will provide OCP evidence of such permission upon OCP’s request. This document and any "Materials" are published on the respective project's wiki page and are open to the public in accordance with OCP's Bylaws and IP Policy. This can be found at

http://www.opencompute.org/participate/legal-documents/. If you have any questions please contact OCP.

Footer:

12. About Open Compute Foundation

The Open Compute Project (OCP) is a collaborative Community of hyperscale data center operators, telecom, colocation providers and enterprise IT users, working with the product and solution vendor ecosystem to develop open innovations deployable from the cloud to the edge. The OCP Foundation is responsible for fostering and serving the OCP Community to meet the market and shape the future, taking hyperscale-led innovations to everyone. Meeting the market is accomplished through addressing challenging market obstacles with open specifications, designs and emerging market programs that showcase OCP-recognized IT equipment and data center facility best practices. Shaping the future includes investing in strategic initiatives and programs that prepare the IT ecosystem for major technology changes, such as AI & ML, optics, advanced cooling techniques, composable memory and silicon. OCP Community-developed open innovations strive to benefit all, optimized through the lens of impact, efficiency, scale and sustainability. Learn more at www.opencompute.org.

Appendix A. Use Cases

1. Inference

1.1. Requirement

Goal: Serve an existing tuned model for a given target, user, application space

Scenario: User wants to connect to a pre-trained/tuned model from a catalog/repository of tuned models for a targeted application domain query interaction, e.g. ChatBot, image caption, genomics query, code segmentation.

- Actor:
  - AI System Engineer
  - Application Engineers
- Preconditions:
  - Data Set, model
    - Location, Metadata, size
    - Type (File, Vector, etc…)
    - Permissions, access
  - System
    - Transaction rate
    - Application target
    - Data store, persistence
  - Method:
    - Attach, service exposure
- Postconditions
  - Model output
  - Metadata
    - Lineage of data
    - Training, tuning info
    - Time stamps
  - Security
    - Access lineage

1.2. No Parallelism

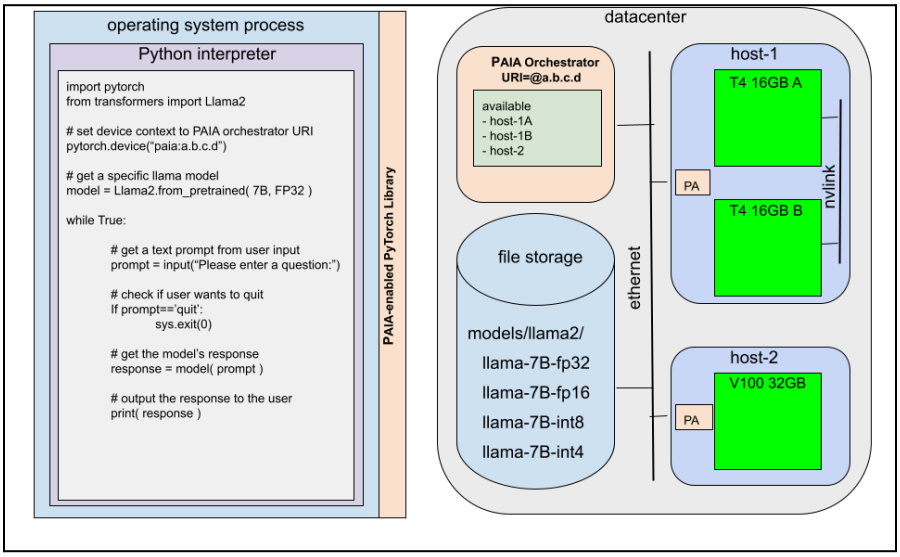
Use the Llama2 model as an example. This family of LLM models has pretrained weights that are openly available from various sources such as HuggingFace:

- Various sizes of models in terms of number of weights
- Different types of model compression schemes such as quantization to low bit fixed point numeric data types

Further, we will explore this use case:

- from the perspective of an AI engineer using a PyTorch library that is PAIA-enabled creating a simple chatbot application
- leverages a fictitious datacenter with a couple of hosts with GPUs attached with specific hardware configurations (this example assumes GPUs with NVLink support)
- The model is able to fit in all the memory of one of the GPUs

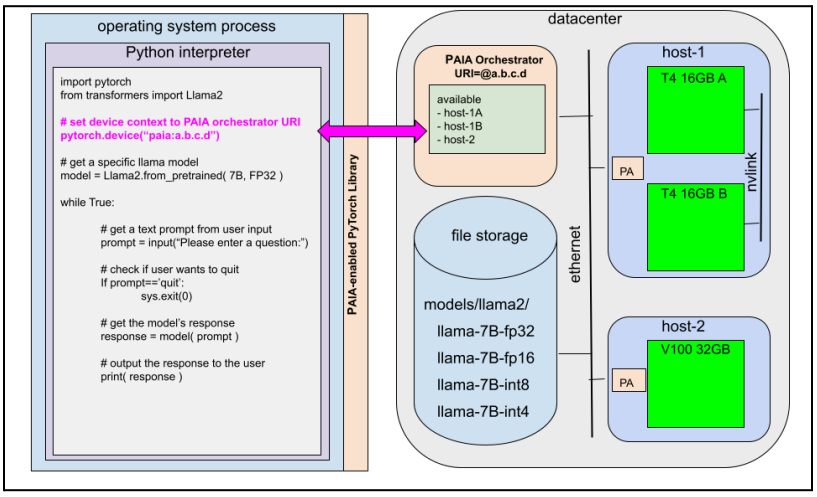
Let's start with a fictitious python chatbot application, built using a PAIA-enabled PyTorch library:



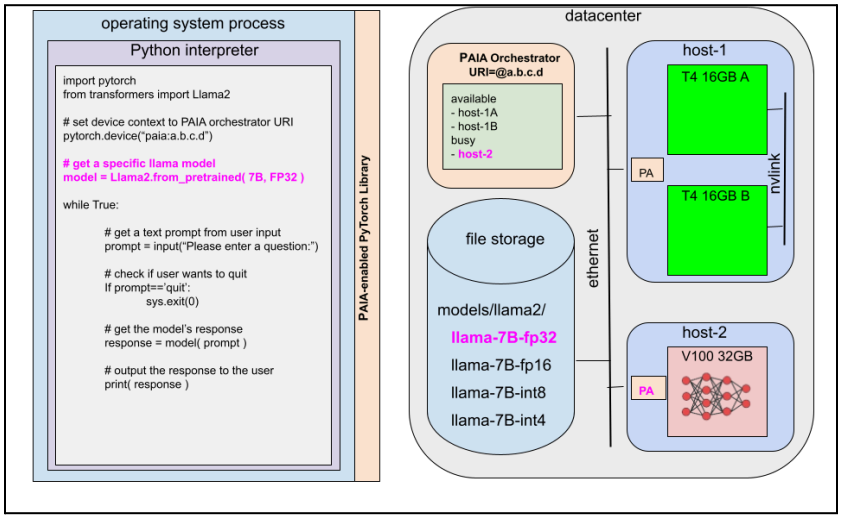
There are a few components worth describing in more details:

- PAIA Orchestrator:
  - Monitors all available resources for availability and capability continuously
  - Makes resource scheduling decisions based on the request and availability
- PAIA Agent:
  - Orchestrator interacts with the compute nodes via this proxy that is generally close to the compute nodes. For example, it could be a small "daemon" that runs in the background of the host.
- PyTorch program
  - It looks like any PyTorch program except the "device context" will be assigned to the datacenter's PAIA Orchestrator
  - The Orchestrator is addressed via some type of URI (ie, ip address, URL, etc.)

Let's now walk through the salient parts of this chatbot program step-by-step.



The highlighted line uses the regular PyTorch syntax to set a device context (typically it might be 'CPU','GPU:0','XLA', etc). This mechanism is how PyTorch can target different accelerator hardware. Here we use it to establish the program's connection to the PAIA-enabled datacenter. All further code will be run in that context. Now, let's load the LLM model.

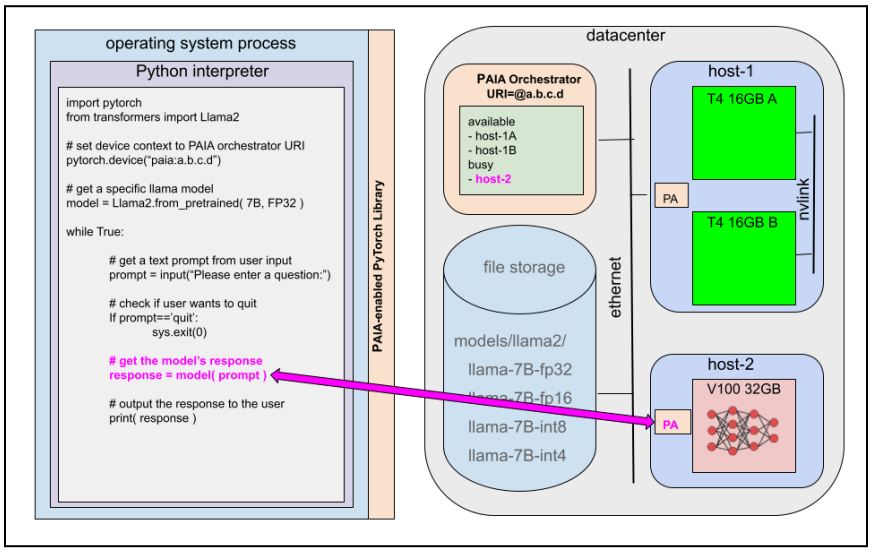


When the highlighted code completes, note the state in the data center:

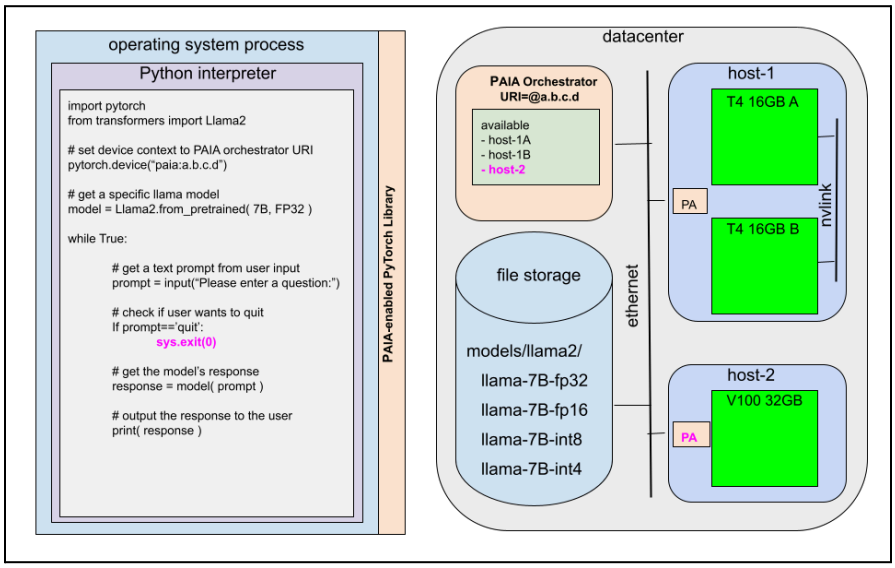
- The Orchestrator found the exact type of LLama model requested (7B weight count and full 32-bit precision) in the datacenter's file storage.
- "host-2" was chosen to run the model and the model's "lowered" compute graph and weights are loaded into the GPU memory by the host's PAIA agent. In the discussion section later, we'll talk about why it chose this resource.
- The Orchestrator marks "host-2" as "busy" in its transactional database.

- A URI of the host is returned as state to the internal session state of the program so that subsequent communication with the model can occur directly and not require the Orchestrator's mediation.

After getting a prompt from the user, the program then executes the model on the input:



Notice here that the program's run-time can communicate directly with the "agent" which was already established when the model was loaded earlier in the program. The Python program continues in a simple while loop until the user enters "quit" and explicitly exits the program:



The shutdown sequence includes:

- Executing a hook installed by the PAIA-enabled PyTorch library, which explicitly tells the host's PA to remove the model from the GPU.
- Re-establish "host-2" GPU's availability.

1.2.1. General Discussion

Obviously, the PAIA Orchestrator and Host Agents are in constant communication via a predefined protocol or (interpreted) execution language (e.g. RESTFUL APIs)]. The communication channels must be highly reliable, resilient, and secure.

Other than providing a PAIA URi device target, the program does not need to explicitly interact with PAIA entities like the Orchestrator or Agent(s). The PAIA-enabled Pytorch run-time maintains the session state implicitly.

The Orchestrator has several responsibilities during inference:

- It needs the ability to locate weights for pre-trained models. In this example, they were available locally, but it may need to download them securely from an external, reputable source. In general, it needs to manage its dedicated file storage efficiently.
- It needs to accurately track the available data center resources continuously and transactionally reserve them for use when they are requested.
- It needs to continuously communicate with an "agent" close to the resources (likely running on each host) for loading/unloading the compute graph and weights, monitoring usage and issues, etc.
- It must decide which resources to allocate based on the requested workload. We discuss its role as a job scheduler next.

1.2.2. Job Scheduling

Let's look at how the Orchestrator decided to use host-1's GPU:

- Orchestrator gets a specific request for Llama2 pre-trained weights, 7B version as well as full precision (FP32). All other model configurations inherit a default value including a *batch size=1*
    - Note that in inference-only mode, the peak activation memory requirements of any pre-trained model can vary depending on how it's configured. For this exercise, we adopt the generally accepted default model configuration (ie. HuggingFace) but let the user program override explicitly.
- Orchestrator computes the estimated amount of peak inference activation memory it will need as follows during decode, which is dominated by the KV cache which is enabled by

default in PyTorch. Let's calculate the max KV cache required for Llama2 at full 32 bit resolution:

- KV cache max = 4096 (context len) X 32 layers X 32 heads X 128 head dims X 2 (ie, K andV) X 4 bytes = 4,294,967,296 = 4GB
- Total Memory (VRAM) requirement = ~28GB (7B weights at full precision) + ~4GB (KV cache at full precision) = ~32GB (it's just a little under)

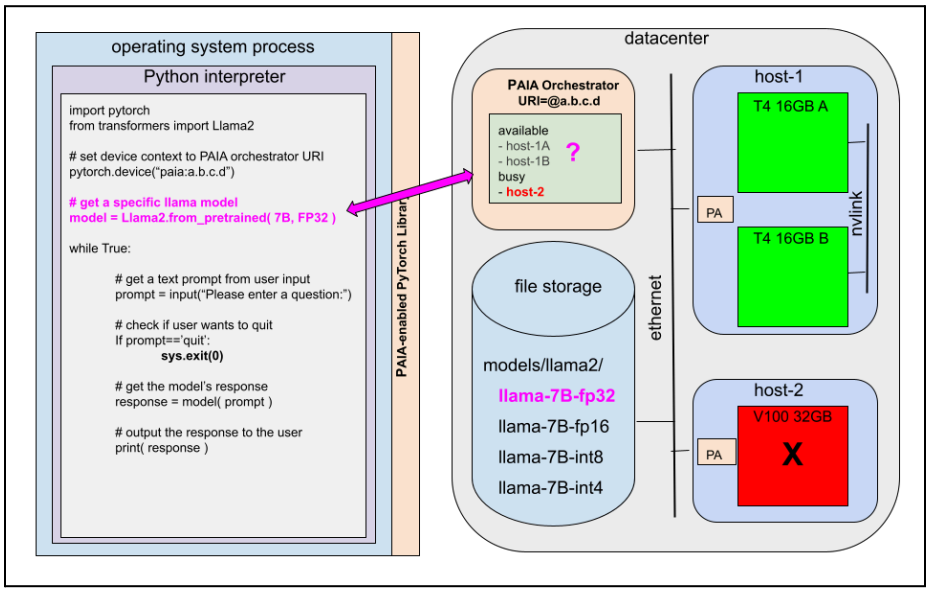
- The Orchestrator locates one compute node which meets the criteria and reserves it for the job.

Notes:

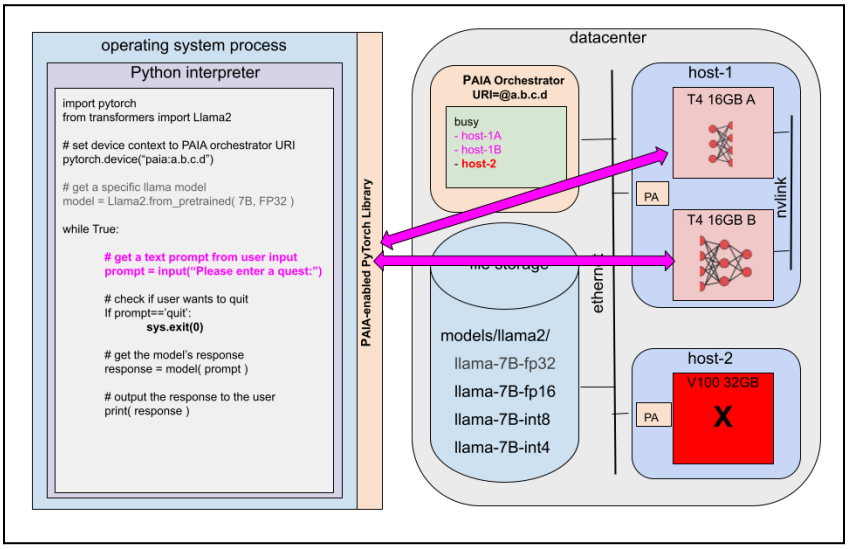
- The system honored the specific model request made by the application, even though "lower resolution" models in the same family appear to be available. If the user does not request specific model configuration(s), we could imagine a system that tries "lower resolution" versions (ie, quantized weights) of the same family until it finds the "next best" fit.
- There are now quite a few techniques to reduce the burden of activation memory size with minimal impact on model accuracy. These are not considered in this exercise, but could be allowed for the automatic configuration of run-time optimizations as long as the decision is transparent to the user.

1.3. Pipeline Parallelism

This is the same as example 1 above, except that the model and its inference activation memory requirements cannot fit into the memory of one of the available GPUs and needs to be split. Let's consider the same chatbot application, but when the Pytorch program requests the model to be loaded, "host-2" is busy and not available:



The Orchestrator decides to split the model and load each partition into the available compute nodes. The application subsequently communicates with the host with the attached compute nodes directly:



1.3.1. Job Scheduling

In this case, the Orchestrator has to make the available resources work. Each available GPU cannot fit the entire model weights and peak inference activation memory, so it can go through a

set of heuristics/decision tree to come up with a solution based on pipeline parallelism, which splits the model along the feed forward direction.

We can simulate the decision process as follows:

- Calculate max VRAM requirement = ~32 GB
- No single compute node can fit it, so try to a split in the middle of the llama transformer at the 16th of 32nd layer and calculate each half's memory requirement:
    - First half including the input: ~14GB weights + ~2GB KV cache = ~16GB
    - Second half (approx the same): ~16GB
    - Each half fits the available compute nodes
- In coordination with the chat app's Pytorch library, the Orchestrator partitions the compute graph appropriately, loads each model partition into the available compute node.

Notes:

- Notice that the Orchestrator makes the decision to partition the model's compute graph, but it needs to coordinate this decision with the Pytorch subsystem at the application side. Pytorch provides powerful abilities that enable arbitrary compute graph transformations at run-time and we envision the Orchestrator leverages this feature.
- Also, by coordinating the partition with the app's Pytorch subsystem, the application then inherits Pytorch's mechanism to manage the data-flow requirements at run-time.
- Other forms of parallelism exist:
    - *Data parallelism* is useful when batch size>1 or in certain streaming use cases not relevant here.
    - *Model parallelism* that splits along internal layer dimensions is also possible, but the heuristic here prefers pipeline parallelism which generally requires less overhead communication between compute node participants and simple pipeline parallelism sufficed in this use case.

2. Pre-Training

Goal: Build a pre-trained model to enable a ChatBot for customer support

Scenario: User wants to pre-train a model from a catalog of targets (eg. GPT-4, Palm, Llama) and parameter sizes to target an application space (Language, Video, Code, Chat, Genomics, etc…) against a dataset.

- Actor: AI System Engineer
- Preconditions:

        - Simulation model outcomes: feedback on resource states
        - Data Set
            - Location
            - Permissions, access
        - System
            - Cost / time to build (not to exceed the $amount or the assigned time using estimated resources)
            - HA, RAS expectation
                - Checkpoints - How often to save model state
        - Model:
            - Target: eg GPT-4
            - Parameter Size
        - Architecture:
            - Transformer
- Postconditions
    - Model output
    - Metadata
        - Lineage of data - tracking data information used in training the model
        - Training info
        - Time stamps
        - Telemetry - performance characteristics
        - Feedback on resource states
    - Security
        - Access

3. Tuning

Goal: Tune an existing pre-trained model for a given target, user, application space

Scenario: User wants to tune a pre-train a model from a catalog of models against a dataset, user space for a targeted application domain

- Actor:
    - AI System Engineer
    - Prompt Engineers
- Preconditions:
    - Data Set
        - Location
        - Type (File, Vector, etc…)
        - Permissions, access

    - System
        - Cost / time to build (not to exceed the $amount or the assigned time using estimated resources)
        - HA, RAS expectation (Checkpoints)
    - Method:
        - RLHF
        - Prompt engineering, Instruct tuning
        - RAG
- Postconditions
    - Model output
    - Metadata
        - Lineage of data
        - Training info
        - Time stamps
        - Model target, tuning
    - Security
        - Access

Appendix B. Compute Graph Operations

We include here a reference for Pytorch's set of compute graph operations via their ATEN intermediate representation. Implementers should consider supporting the following compute graph operators to be complete, although the Pytorch ecosystem has been moving aggressively towards smaller canonical compute graph operators including Core ATEN IR, PRIM IR, and more recently XLA's operator set.

name	description	Pytorch documentation
abs_tensor*	take element-wise absolute value of a tensor	https://pytorch.org/docs/stable/generated/torch.abs.html
acos	inverse cosine operation	https://pytorch.org/docs/stable/generated/torch.acos.html
acosh	inverse cosine hyperbolic	https://pytorch.org/docs/stable/generated/torch.acosh.html
adaptive_avg_pool1d	pooling operation with overlap reduction	https://pytorch.org/docs/stable/generated/torch.nn.functional.adaptive_avg_pool1d.html

adaptive_avg_pool2d	pooling operation with overlap reduction	https://pytorch.org/docs/stable/generated/torch.nn.AdaptiveAvgPool2d.html
adaptive_max_pool3d	pooling operation with overlap reduction	https://pytorch.org/docs/stable/generated/torch.nn.AdaptiveAvgPool3d.html
add_scalar*	basic tensor operation adding scalar value to tensor	https://pytorch.org/docs/stable/generated/torch.Tensor.add.html
add_tensor*	basic tensor operation adding tensors	https://pytorch.org/docs/stable/generated/torch.add.html
addmm	Matmul of two tensors followed an add	https://pytorch.org/docs/stable/generated/torch.addmm.html
alias	Create an alias for a tensor	?
amax	Return max value of tensor in a dim	https://pytorch.org/docs/stable/generated/torch.amax.html
amin	Return max value of tensor in a dim	https://pytorch.org/docs/stable/generated/torch.amin.html
any	Similar to numpy any	https://pytorch.org/docs/stable/generated/torch.any.html
any_dim*	Same as “any” but with explicit dim	https://pytorch.org/docs/stable/generated/torch.any.html
any_dim2*	Same as “any_dim” but with explicit multi-dim	https://pytorch.org/docs/stable/generated/torch.any.html
arange_start_step*	Similar to numpy arange	https://discuss.pytorch.org/t/torch-arange-in-start-end/33258
argmax	Get tensor dim of max value	https://pytorch.org/docs/stable/generated/torch.argmax.html
argmin	Get tensor dim of min value	https://pytorch.org/docs/stable/generated/torch.argmin.html
as_strided	Create a view of a tensor with offset and stride	https://pytorch.org/docs/stable/generated/torch.as_strided.html

asin	in Yu verse sin	https://pytorch.org/docs/stable/generated/torch.asin.html
asinh	inverse sin hyperbolic	https://pytorch.org/docs/stable/generated/torch.asinh.html
atan	Inverse tangent	https://pytorch.org/docs/stable/generated/torch.atan.html
atan2	Inverse tangent that considers quadrant	https://pytorch.org/docs/stable/generated/torch.atan2.html
avg_pool1d	1d pooling operation with averaging	https://pytorch.org/docs/stable/generated/torch.nn.functional.avg_pool1d.html
avg_pool2d	2d pooling operation with averaging	https://pytorch.org/docs/stable/generated/torch.nn.functional.avg_pool2d.html
avg_poold	3d pooling operation with averaging	https://pytorch.org/docs/stable/generated/torch.nn.functional.avg_pool3d.html
batchnorm2d*	batch normalization	https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html
bitwise_and_scalar*	Basic tensor scalar bitwise operation	https://pytorch.org/docs/stable/generated/torch.bitwise_and.html
bitwise_and_tensor*	Basic tensor tensor bitwise operation	https://pytorch.org/docs/stable/generated/torch.bitwise_and.html
bitwise_or_scalar*	Basic tensor scalar bitwise operation	https://pytorch.org/docs/stable/generated/torch.bitwise_or.html
bitwise_or_tensor*	Basic tensor tensor bitwise operation	https://pytorch.org/docs/stable/generated/torch.bitwise_or.html
bitwise_xor_scalar*	Basic tensor scalar bitwise operation	https://pytorch.org/docs/stable/generated/torch.bitwise_xor.html
bitwise_xor_tensor*	Basic tensor tensor bitwise operation	https://pytorch.org/docs/stable/generated/torch.bitwise_xor.html
bmm	Batch matmul	https://pytorch.org/docs/stable/generated/torch.bmm.html
cat	Concat tensors along a given dim	https://pytorch.org/docs/stable/generated/torch.cat.html

ceil	Returns ceiling of tensor	https://pytorch.org/docs/stable/generated/torch.ceil.html
clamp	Clamp elements of tensor	https://pytorch.org/docs/stable/generated/torch.clamp.html
clone	Deep clone a tensor	https://pytorch.org/docs/stable/generated/torch.clone.html
cdist*	pnorm distance	https://pytorch.org/docs/stable/generated/torch.cdist.html
col2im	Rearrange columns of a tensor	https://pytorch.org/cppdocs/api/function_namespaceat_1a979fbf85d8c7362d60d766bbf1639f10.html
constant_pad_nd*	Pad a tensor along n dimensions with a value	https://pytorch.org/docs/stable/generated/torch.nn.ConstantPad2d.html
convolution	Pytorch convolution operator	Various layers from: https://pytorch.org/docs/stable/nn.html#convolution-layers
copy_tensor*	Basic tensor copy operation	https://pytorch.org/cppdocs/api/function_namespacetorch_1ad096cf1dbb94e3ec63b75e7587e1dddb.html
cos	cosine	https://pytorch.org/docs/stable/generated/torch.cos.html
cosh	cosine hyperbolic	https://pytorch.org/docs/stable/generated/torch.cosh.html
cumsum	cumulative sum along dim of a tensor	https://pytorch.org/docs/stable/generated/torch.cumsum.html
diagonal	Partial view of a tensor with diagonal appended	https://pytorch.org/docs/stable/generated/torch.diagonal.html
div_scalar*	Divide a tensor by a scalar	https://pytorch.org/docs/stable/generated/torch.div.html
div_scalar_mode*	Divide a tensor by a scalar with rounding mode	https://pytorch.org/docs/stable/generated/torch.div.html
div_tensor*	Divide a tensor by a tensor	https://pytorch.org/docs/stable/generated/to

		rch.div.html
embedding	Embedding lookup table	https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html
empty_memory_format*	Create empty tensor with memory format requirement	https://pytorch.org/docs/stable/generated/torch.empty.html
empty_strided	Create empty tensor with stride requirement	https://pytorch.org/docs/stable/generated/torch.empty.html
equality_scalar*	Tensor equality against scalar via broadcast	https://pytorch.org/docs/stable/generated/torch.eq.html
equality_tensor*	Tensor equality against tensor	https://pytorch.org/docs/stable/generated/torch.eq.html
erf	Compute error function of input	https://pytorch.org/docs/stable/special.html#torch.special.erf
exp	Return element wise exponential of the input	https://pytorch.org/docs/stable/generated/torch.exp.html
expand	Expand a tensor	https://pytorch.org/docs/stable/generated/torch.Tensor.expand.html
expm1	Return element wise exponential of the input minus 1	https://pytorch.org/docs/stable/special.html#torch.special.expm1
fill_scalar*	Creates tensor with value to fill	https://pytorch.org/docs/stable/generated/torch.full.html
flip	Reverse order of tensor along a dim	https://pytorch.org/docs/stable/generated/torch.flip.html
floor	Returns floor of tensor	https://pytorch.org/docs/stable/generated/torch.floor.html
fmod_scalar*	Element-wise application of fmod to a tensor with a scalar	https://pytorch.org/docs/stable/generated/torch.fmod.html
fmod_tensor*	Element-wise application of fmod to a tensor with a	https://pytorch.org/docs/stable/generated/torch.fmod.html

	tensor	
full	Create tensor with fill value, type inferred	https://pytorch.org/docs/stable/generated/torch.full.html
gather	Gather values along a tensor dim	https://pytorch.org/docs/stable/generated/torch.gather.html
ge_scalar*	Tensor element-wise greater than or equal to a scalar	https://pytorch.org/docs/stable/generated/torch.ge.html
ge_vector*	Tensor element-wise greater than or equal to another tensor	https://pytorch.org/docs/stable/generated/torch.ge.html
gelu	Gaussian error-based activation	https://pytorch.org/docs/stable/generated/torch.nn.functional.gelu.html
grid_sampler_2d	Image sampling	https://pytorch.org/docs/stable/generated/torch.nn.functional.grid_sample.html
gt_scalar*	Tensor element-wise greater than a scalar	https://pytorch.org/docs/stable/generated/torch.gt.html
gt_vector*	Tensor element-wise greater than another tensor	https://pytorch.org/docs/stable/generated/torch.gt.html
hardtanh	Element-wise hyperbolic tangent with min/max	https://pytorch.org/docs/stable/generated/torch.nn.Hardtanh.html
index	Tensor getter indexing (normal)	https://pytorch.org/cppdocs/notes/tensor_indexing.html
index_put	Tensor setter indexing	https://pytorch.org/cppdocs/notes/tensor_indexing.html
index_select	Tensor getter indexing by tensor	https://pytorch.org/docs/stable/generated/torch.index_select.html
isinf	Element-wise infinite check (neg or pos)	https://pytorch.org/docs/stable/generated/torch.isinf.html
isnan	Element-wise not-a-number check (neg or pos)	https://pytorch.org/docs/stable/generated/torch.isnan.html

le_scalar*	Tensor element-wise less than or equal to a scalar	https://pytorch.org/docs/stable/generated/torch.le.html
le_vector*	Tensor element-wise less than or equal to another tensor	https://pytorch.org/docs/stable/generated/torch.le.html
leaky_relu	Leaky version of RELU activation	https://pytorch.org/docs/stable/generated/torch.nn.functional.leaky_relu.html
log	Element-wise natural logarithm	https://pytorch.org/docs/stable/generated/torch.log.html
log10	Element-wise logarithm base 10	https://pytorch.org/docs/stable/generated/torch.log10.html
log1p	Element-wise natural logarithm, adding 1 to input	https://pytorch.org/docs/stable/generated/torch.log1p.html
log2	Element-wise logarithm base 2	https://pytorch.org/docs/stable/generated/torch.log2.html
logical_and	Element-wise logical AND of two tensors	https://pytorch.org/docs/stable/generated/torch.logical_and.html
logical_not	Element-wise logical NOT	https://pytorch.org/docs/stable/generated/torch.logical_not.html
logical_or	Element-wise logical OR of two tensors	https://pytorch.org/docs/stable/generated/torch.logical_or.html
logical_xor	Element-wise logical XOR of two tensors	https://pytorch.org/docs/stable/generated/torch.logical_xor.html
log_softmax	Log version of softmax	https://pytorch.org/docs/stable/generated/torch.nn.LogSoftmax.html
lt_scalar*	Tensor element-wise less than a scalar	https://pytorch.org/docs/stable/generated/torch.lt.html
lt_vector*	Tensor element-wise less than another tensor	https://pytorch.org/docs/stable/generated/torch.lt.html
max_dim*	Max of Tensor along a dim	https://pytorch.org/docs/stable/generated/torch.max.html
max_pool2d_with_indi	Max pooling for 2D, return	https://pytorch.org/cppdocs/api/function_na

ces*	indices	mespacetorch_1_1nn_1_1functional_1a312c870582822171c427b5713f9c9bf9.html
max_pool3d_with_indices*	Max pooling for 3D, return indices	https://pytorch.org/cppdocs/api/function_namespacetorch_1_1nn_1_1functional_1a3fee57db9a7ca3969e7d6299a0b1a660.html
maximum	Element-wise max between two tensors	https://pytorch.org/docs/stable/generated/torch.maximum.html
mean	Return mean of tensor along dim	https://pytorch.org/docs/stable/generated/torch.mean.html
min	Return min of tensor along dim	https://pytorch.org/docs/stable/generated/torch.min.html
minimum	Element-wise min between two tensors	https://pytorch.org/docs/stable/generated/torch.minimum.html
mm	matmul	https://pytorch.org/docs/stable/generated/torch.mm.html
mul_scalar*	Element-wise mult	https://pytorch.org/docs/stable/generated/torch.mul.html
mul_tensor*	Tensor mult with broadcast	https://pytorch.org/docs/stable/generated/torch.mul.html
native_dropout*	Dropout (normal)	https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html
native_groupnorm*	Group normalization over a mini-batch	https://pytorch.org/docs/stable/generated/torch.nn.GroupNorm.html
native_layernorm*	Layer normalization	https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html
ne_scalar*	Tensor element-wise not equal to a scalar	https://pytorch.org/docs/stable/generated/torch.ne.html
ne_vector*	Tensor element-wise not equal to a tensor	https://pytorch.org/docs/stable/generated/torch.ne.html
negative	Negative of a tensor	https://pytorch.org/docs/stable/generated/torch.negative.html
nonzero	Return indices of nonzero	https://pytorch.org/docs/stable/generated/to

	elements	rch.nonzero.html
permute	View of tensor with indices permuted	https://pytorch.org/docs/stable/generated/torch.permute.html
pixel_shuffle*	Rearrange tensor elements with upscale factor	https://pytorch.org/docs/stable/generated/torch.nn.PixelShuffle.html
pow_scalar*	Element-wise power of tensor	https://pytorch.org/docs/stable/generated/torch.pow.html
pow_tensor*	Element-wise power of tensor using another tensor	https://pytorch.org/docs/stable/generated/torch.pow.html
product	Product of tensor elements	https://pytorch.org/docs/stable/generated/torch.prod.html
product_dim*	Product of tensor elements along dim	https://pytorch.org/docs/stable/generated/torch.prod.html
rand	Create tensor with random elements	https://pytorch.org/docs/stable/generated/torch.rand.html
randn	Create tensor with random elements using normal distribution	https://pytorch.org/docs/stable/generated/torch.randn.html
randperm	Create tensor with random elements between a range	https://pytorch.org/docs/stable/generated/torch.randperm.html
reciprocal	Element-wise reciprocal	https://pytorch.org/docs/stable/generated/torch.reciprocal.html
reflection_pad1d*	Pad tensor by boundary reflection	https://pytorch.org/docs/stable/generated/torch.nn.ReflectionPad1d.html
reflection_pad2d*	Pad tensor by boundary reflection	https://pytorch.org/docs/stable/generated/torch.nn.ReflectionPad2d.html
reflection_pad3d*	Pad tensor by boundary reflection	https://pytorch.org/docs/stable/generated/torch.nn.ReflectionPad3d.html
relu	RELU activation	https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html
remainder_scalar*	Element-wise modulus	https://pytorch.org/docs/stable/generated/torch.remainder.html

remainder_tensor*	Element-wise modulus with tensor	https://pytorch.org/docs/stable/generated/torch.remainder.html
repeat	Repeat tensor along dim(s)	https://pytorch.org/docs/stable/generated/torch.Tensor.repeat.html
replication_pad2d*	Pad tensor by boundary replication	https://pytorch.org/docs/stable/generated/torch.nn.ReplicationPad2d.html
replication_pad3d*	Pad tensor by boundary replication	https://pytorch.org/docs/stable/generated/torch.nn.ReplicationPad3d.html
resize	Resize tensor	https://pytorch.org/docs/stable/generated/torch.Tensor.resize_.html
roll	Element-wise shift of tensor	https://pytorch.org/docs/stable/generated/torch.roll.html
round	Round tensor elements	https://pytorch.org/docs/stable/generated/torch.round.html
rsqrt	Element-wise reciprocal of square root	https://pytorch.org/docs/stable/generated/torch.rsqrt.html
scalar_tensor*	A scalar (constant) tensor	https://pytorch.org/docs/stable/tensors.html
scatter_src*	Scatter-based reduction from a source Tensor	https://pytorch.org/docs/stable/generated/torch.Tensor.scatter_.html#torch.Tensor.scatter_
scatter_value*	TBD	https://pytorch.org/docs/stable/generated/torch.Tensor.scatter_.html#torch.Tensor.scatter_
scatter_add*	TBD	https://pytorch.org/docs/stable/generated/torch.Tensor.scatter_.html#torch.Tensor.scatter_
scatter_reduce_two*	TBD	https://pytorch.org/docs/stable/generated/torch.Tensor.scatter_.html#torch.Tensor.scatter_
sigmoid	Sigmoid activation	https://pytorch.org/docs/stable/generated/torch.nn.Sigmoid.html
sign	Element-wise sign (-1,0,1)	https://pytorch.org/docs/stable/generated/to

	of a tensor	rch.sign.html
sin	Element-wise sin of a tensor	https://pytorch.org/docs/stable/generated/torch.sin.html
sinh	Element-wise hyperbolic sin of a tensor	https://pytorch.org/docs/stable/generated/torch.sinh.html
slice_tensor*	Tensor indexing	https://pytorch.org/cppdocs/notes/tensor_indexing.html
slice_scatter*	TBD	https://pytorch.org/docs/stable/generated/torch.slice_scatter.html
sort	Sort Tensor along dims	https://pytorch.org/docs/stable/generated/torch.sort.html
split_with_sizes*	TBD	TBD
sqrt	Element-wise sqrt of a tensor	https://pytorch.org/docs/stable/generated/torch.sqrt.html
squeeze_dim*	TBD	TBD
squeeze_dims*	TBD	TBD
sub_scalar*	TBD	TBD
sub_tensor*	TBD	TBD
sum_dim*	Element-size sum along dim	https://pytorch.org/docs/stable/generated/torch.sum.html
sym_humel*	TBD	TBD
sym_size*	TBD	TBD
sym_storage_offset*	TBD	TBD
sym_stride*	TBD	TBD
tan	Element-size tangent	https://pytorch.org/docs/stable/generated/torch.tan.html
tanh	Element-size hyperbolic	https://pytorch.org/docs/stable/generated/to

	tangent	rch.tanh.html
topk	Returns k largest along a dim	https://pytorch.org/docs/stable/generated/torch.topk.html
trunc	Element-size integer truncation of a tensor	https://pytorch.org/docs/stable/generated/torch.trunc.html
unsqueeze	TBD	TBD
upsample_bilinear2d	Upsample 2d tensor using bilinear interpolation	https://pytorch.org/docs/stable/generated/torch.nn.Upsample.html
upsample_nearest2d	Upsample 2d tensor using nearest interpolation	https://pytorch.org/docs/stable/generated/torch.nn.Upsample.html
var_correction*	Compute variance with correction	https://pytorch.org/docs/stable/generated/torch.var.html
var_dim*	Compute variance along a dim	https://pytorch.org/docs/stable/generated/torch.var.html
view	Create a tensor view	https://pytorch.org/docs/stable/tensor_view.html
where	Tensor selection via a conditional	https://pytorch.org/docs/stable/generated/torch.where.html

The following Pytorch ATEN IR operators were not (yet) considered:

- _adaptive_avg_pool2d_backward
- _native_batch_norm_legit
- _native_batch_norm_legit.no_stats
- _native_batch_norm_legit_no_training
- _to_copy
- atan2.out
- avg_pool2d_backward
- embedding_dense_backward
- local_scalar _dense
- max_2d_with_indices_backward
- max_pool2d_with_indices_backward
- native_groupnorm_backward
- native_layernorm_backward

1. Fixed Point Types

Note that we only consider common fixed point integer formats, so no bits are used for fractional bits. These are pretty standard data types and likely represent the super set that we might care about.

Name	Storage (bits)	Format	Numeric Range (inclusive)	Notes
UINT4	4	unsigned	0 to 2^4-1	
INT4	4	signed	-2^(4-1) to 2^(4-1)-1	
UINT8	8	unsigned	0 to 2^8-1	
INT8	8	signed	-2^(8-1) to 2^(8-1)-1	
UINT16	16	unsigned	0 to 2^16-1	
INT16	16	signed	-2^(16-1) to 2^(16-1)-1	
UINT32	32	unsigned	0 to 2^32-1	
INT32	32	signed	-2^(32-1) to 2^(32-1)-1	
UINT64	64	unsigned	0 to 2^64-1	
INT64	64	signed	-2^(64-1) to 2^(64-1)-1	

2. Floating Point Types

We support the common IEEE754 half, single, and double precision floating point formats but include newer formats targeted specifically for machine/deep learning.

Name	Storage (bits)	Format (S/E/M)	Owner/Standard	Notes	Ref
f8E4M3FN	8	1/4/3	NV	INF, -INF not rep	link
f8E5M2	8	1/5/2	NV	INF, -INF, NaN rep	link

f8E4M3FNUZ	8	1/4/3	Graphcore		link
f8E5M2FNUZ	8	1/5/2	Graphcore		link
f8E4M3B11FNUZ	8	1/4/3	IBM		link
FP16	16	1/5/10	IEEE754 (hp)		link
BFP16	16	1/8/7	Google	Google “B”rain format	link
TF32	19	1/8/10	NV	Developed for NV A100	link
FP32	32	1/8/23	IEEE754 (sp)		link
FP64	64	1/11/52	IEEE754 (dp)		link

The following floating point formats are not (yet) considered:

- Decimal (from IEEE754)
- MSFP8, MSFP9 (from Microsoft Brain Wave)
- FlexPoint (from Nirvana)
- Posit/Unum

Appendix C. Security and Threats

Principles of Zero Trust revolutionizes cyber defenses by replacing the core security principles. Any client, regardless of whether it is in an internal or external network, is considered a suspect that may be used to commence attacks against the organization.

- Assume breach
- Never trust, always verify
    - Eliminate implicit trust
    - Minimize explicit trust
    - Continually monitor and apply dynamic policy to verify trustworthiness

To software developers and security professionals, hardware and firmware are typically out of scope and treated as black box. Software/Hardware Co-Design makes it possible to achieve the goal of zero trust security principle.

Machine learning based software development workflow includes three main artifacts: Data, AI/ML Model, and Code. There are three types of risk factors related to typical machine learning workflow:

- AI Data Risk
- AI Model Risk
- AI Code Risk

In the realm of AI/ML system threat modeling, it is necessary to broaden the focus beyond the intricacies of code and traditional database/information security and extend it to encompass the new dimensions of AI/ML data and model security.

In training large models for deployment, e.g. LLMs, there are a number of actors performing actions on data over a span of time to produce a result. To efficiently compute on the data optimizations are performed in how data is stored, transmitted and used. Through this process large data sets are processed to produce a much smaller result, accurate representation of the data. This compression flow can be subject to data manipulated along the process to possibly influence the model output. To protect against this there are modes of protection for

- Data at Rest - encryption
- Data in Transit
- Data in Use - confidential computing

Raw data sources are processed, classified, tokenized to produce a training data set fit to purpose. As with the computation model for training the security of the production of the initial data set needs to be managed.

During training the data may be split across one or more pipelines, communicated from durable stores such as a Parallel File System or Object Store to compute nodes. This communication needs to manage the batch process for correct split as well the integrity of the data to be processed. Once data has landed the computation cycle produces intermediate results that may not fit in the computational element. For this additional compute and memory sources are required. This increases the size of the computation security enclave for this phase of processing. Connecting these phases into a pipeline allows for scaling computation but also adds a communication factor for managing parameter updates, state, in building the best end results.

Possible mitigation techniques to explore

- Orchestrating, attesting firmware/software versions used in the overall cycle
    - Supply chain to deployment
- Encrypting Links for communicating traffic
- Versioning, signing data
    - Training
    - Model outputs
- Lineage
    - Data used for training to model produced

To simplify the threat modeling process, it might be helpful to perform threat modeling on each of the AI Risk factors (Data, Model, Code) as part of independent workflows. In this specification, we define some simple attributes on the risk factors to indicate if compute execution is trustable.

- Trustability: [yes|no] - for all Polymorphic Architecture components, compute, interconnect, memory, etc.
- Scaled-up or scale-out unit forms a secure enclave, if and only if the individual compute/interconnect/memory component is secure and input data is trustable.

The management APIs to enter/exit the Trusted Execution Environment (TEE):

- check - validate individual architecture component
- de/encrypt - data encryption and decryption
- export - transport the data from secure enclave to regular execution environment

Currently there is a lot of debate on the licensing and free use of data for training models. For this reason, governance is out of scope in this revision as well as the mapping model built to compliance such as GDPR.

Appendix D. A sample implementation of PolyArch

1. Definitions of key software elements for simulation

```
def get_devices(n):
    device set <= make virtual device set;
    return device set

a <= map( lambda func1, get_devices(#) )
b <= map( lambda func2, get_devices(#) )
c <= map( lambda func3, get_devices(#) )

def f (var):
    x <= a(var)
    y <= b(x)
    z <= a(b(x))
    return (y, z)
```

Program: Similar to a Google Pathways program [3]. It is comprised of three parts:

- Specification of *kernels* made of PolyArch composable operators
- Agreement to allocate a number of each type of kernel.
- Actual program which specifies how the kernels interact and pass data to each other. Programs are represented as direct acyclic graphs (DAGs), which map the topology of interconnected resource hierarchies for the program.

PolyArch operator language: a domain-specific language (DSL) to specify kernels operators and a type system to express allowed types, but also addresses extensibility. In the above example, the lambda form is the Google Pathways operator language.

Kernel: a set of operations expressed in the PolyArch operator language which the program composes. In the above example, a, b, and c are the kernels.

Program Compiler: The program compiler takes the specified kernel operators and checks to see if the hardware on the system can support the operators which the kernels use as well as the availability of logical composed resources on the system.

AI accelerator: an AI accelerator is any hardware which can potentially execute an operator which a kernel in a program may require. This may be an AI accelerator, GPU, FPGA, or CPU. If

a kernel specifies an operation which is not supported by all the AI accelerators in the PolyArch system, the program compiler will return an error indicating the operation is unsupported.

As a part of integrating the AI accelerator hardware into a PolyArch system, each implementer of a hardware accelerator needs to add the ability to compile an PolyArch operator into the native hardware code or executable representation for the accelerator.

If no AI accelerator can support a specific operator, if the system and a CPU, the system could compile a set of PolyArch operators supported by the lower hierarchy of computing resources.

1.1. Terms borrowed from Google Pathways [3]

Island: logical grouping of AI hardware allocated from the datacenter. In PolyArch, an island can be further scoped out to include all discovered resources tagged to the island. All resources are identified by their GUIDs. Programs allocate a group of kernels to map an island (programs could have an implicit default island which the program's execution environment defines). Islands may be homogenous (e.g., all comprise one type of AI accelerator) or heterogeneous (composed of multiple AI accelerator types).

The PolyArch architecture also extends the notion of an island to be the unit of hardware provisioned for a given program for different kinds of resources. The PolyArch supports three types of islands. The interconnect capability between island resources are described via the definitions in Chapter 3.

- AI Accelerator Islands
- Memory Islands
- Storage Islands

The architecture expects that islands may have different sizes, compute/storage capabilities, costs, and quality of services.

The size of an island is variable with application, and may map to a variety of things that can be orchestrated: routines at linker/library level, programs, containers or virtual machines or specialized machines.

AI Accelerator Islands: This type of island is similar to the island described in the Google Pathways, but requires heterogeneous accelerator support for the PolyArch. Each island is associated with a description of what PolyArch operators the hardware supports. The (dynamic) program compiler uses this descriptor to generate code to target an island with different kinds of AI accelerators. Multiple kinds of AI accelerators could be composed to implement the same operator.

Each AI accelerator island also has its software components to support hardware-specific optimization and execution: a scheduler, a compiler (or compilers) which can generate the hardware operations to implement a kernel for the hardware in the island, and management context.

Memory and Storage Islands

Though Google Pathways specifies names for logical memory allocations like the sharded buffers, the PolyArch anticipates the need of full descriptions of interconnect bandwidths and topology for disaggregated memory and storage. These resources could be used for programs to communicate with intents across islands, save checkpoints, persist results past the lifetime of the program for future use, etc. (In this case, the DAG compute elements need to be augmented with additional information such as checkpoints, etc.) Similarly, these could also be used for the memory and storage objects which must be shared across kernels in different islands.

The memory and storage islands will have specific components and based on those components, have specific characteristics. System designers can decide whether all the components in a memory island are of the same type/capability of component (e.g., HBM, DDR-DRAM, CXL, etc) and, similarly for storage islands whether they are NVMe, SATA, or more traditional NFS or SAN storage devices.

For shared memory buffers and storage shared between programs, we can allow the creation of an explicit namespace (mapped to GUID in the PolyArch control plane), for which different program instances (perhaps run from different users) may access object names

We recognize there may be memory and storage devices within an AI accelerator island, but these devices are described as shareable or unshareable (utilized implicitly in the island runtime environment) as shown in Section 4.2. The unused resources on the island-local memory and storage devices are considered stranded – not shareable with other islands - unless there are remaining interconnect bandwidths for them to connect to other islands or nodes. In this case, the stranded resources in this island can be reused. For disaggregated media, this design assumes, like AI Accelerator islands, that in production, there will be few configurations for disaggregated memory and storage.

For each disaggregated media island type, the system will also provide the ability to compute the performance and cost/quality metrics between an AI accelerator island and memory and storage island. This might be used in a global scheduler resource for co-location during the running of a program or a way to gauge the memory latency to a memory island.

We do not discuss the namespace rules for the objects which may have been accessed. It is likely the model follows similar rules to Google Pathways naming for buffer objects, accounting

for kernels in a computation to share buffer objects, perhaps via RDMA across islands, as well as the use case where multiple programs may want to access buffer or storage objects.

Compute Interconnect: PolyArch considers this as a type of resource to be allocated for both intra-island and inter-island accelerators to communicate. The physical topology is left to be defined, but it should be flexible enough to allow for the communication patterns and require (allow for 2D, 3D, mesh or other topologies) . This is equivalent to the Google Pathways term ICI.

Datacenter Network (DCN): Infrastructure for islands to communicate, e.g., if islands are in separate datacenter racks.

1.2. Additional System Considerations

AI System Infrastructure: There are other compute-related resources, which are not visible to the program but helps with dispatch of work, management of the hardware, telemetry, and SLAAdvertised capacity for a given capability. The infrastructure monitors and tracks currently running program instances and their overall utilization of PolyArch hardware resources.

2. PolyArch operator language specification

PolyArch expects there exists vendor-specific low-level software support to operate on heterogeneous hardware. Thus, the PolyArch defines the abstract operations and data types which can be mapped to the underlying resources. The PolyArch runtime kernels may be implemented in different kinds of hardware,

- Scalars to elements of aggregate types
- Aggregate type to aggregate type operators
- Function to apply to elements of an aggregate type. It is assumed the lambda is applied to a scalar value. The lambda also specifies a predicate which, if **TRUE**, the lambda is applied to the element, as well as a value to return if the predicate is **FALSE**. Note: that the term lambda is used in a different context from the lambda used in the specification of a kernel in the Google Pathways paper.

PolyArch operator language specification

The implementation of PolyArch operator language can be very detailed as defined in Appendix for Compute Graph Operations. However, for a high-level cycle-approximate simulation, the

abstract operator categories and data type templates can be used to reduce the simulation time, especially when the simulated system is at scale.

Either detailed or abstract, the BNF grammars for valid types and operations of the PolyArch operator language can be specified accordingly.

3. Using the System

After the system is started and fully configured, all available resources are registered to the inventory of the PolyArch resource manager, so that programs can make requests to allocate and utilize them. Each island type, if available in the system, requires an GUID (or simply ***island type ID*** as in [3]) which describes the hardware system.

- ***AI Accelerator island type ID***:
- ***Memory island type ID***: Unit of memory resource provisioning of a specific kind of memory.
- ***Storage island type ID***:

Each of the island type IDs is associated with information about the list of components and their properties. It is assumed a PolyArch system will have at least one ***AI Accelerator type ID***.

Compiling a program

Each program will specify the use of kernels in a manner similar to Google Pathways. However, because the PolyArch will be supporting a heterogenous set of AI accelerators, the program compiler or the programmer will need to determine whether an island supports the operations a kernel requires.

To determine if a kernel is implementable, PolyArch specifies every island is associated with an ***AI accelerator island type ID***. Associated with each accelerator island type, the PolyArch system maintains a description of the set of PolyArch operators an island supports, the AI accelerators available within this type of island, and which AI accelerator implements which operators. The program compiler checks to see if the default accelerator island type associated with the user can support the specified kernels. If it can't, the programmer may be able to use facilities to query other island types to see if one of them supports a specific kernel.

The successful compilation of the above get_devices() indicates the island type supports the above kernels, but makes no guarantees about whether the kernels themselves can be allocated at runtime.

The expectation is that for the most part, there will be a few island types of disaggregated resources (accelerators, operation modes, or by data types) in a production system. For groups

exploring new AI hardware accelerator capabilities or doing compatibility testing, their PolyArch systems may require many different island types.

If a program needs to access disaggregated memory or storage resources, it needs to access them from a media island with a specific ***memory island type ID*** or a ***storage island type ID***.

Resource allocation while running a program

At runtime, each instance of the program run creates a unique global id for the instance of the program being run: ***program instance id***. This is the root context for which the state of the overall computation and the allocated hardware resources (via hierarchies of GUIDs that describe resource mappings to app/program as well as logs, telemetry, etc.) are associated to run this program.

Each program request allocates the kernels from the island's resource manager to support the program. The program will get an ***AI accelerator island chunk ID***, which the island assigns on allocation, for each set of kernels it requests from the island. This corresponds to the use of AI accelerator hardware associated with this program. The context associated with the program instance id will maintain information about which kernels are using a specific ***AI accelerator island chunk ID***. The island will maintain information about which ***program instance ID*** is associated with each AIA island chunk id.

For the storage islands, the corresponding object name and namespace path provides the equivalent of an AI accelerator island chunk id. If a program accesses or allocates storage objects to each storage island from which the program is using a specific named object on a path, the storage island maintains the list of accessing program ***instance IDs*** which are accessing the storage object named object and path.

[note: whether or not two programs can share island instance ids depends on the island tenancy, app attributes / properties, and the PolyArch security policy.]

AI Accelerator Island: internal hardware and software components

One of the differences between the Google Pathways runtime and the PolyArch architecture is the underlying compute hierarchy is not necessarily homogeneous and may not be uniform across all physical racks (or hierarchically discovered resources across an interconnect topology) in the datacenter. This model allows the datacenter infrastructure team to either disaggregate similar compute types, e.g, have Accelerator island types which are homogenous – all have mostly the same type of AI accelerator hardware, with the exception of a couple of general purpose CPUs, or allow AI accelerator islands to have a mix of AI accelerator hardware types.

Island compute HW support

For each ***accelerator island type ID***, the system maintains the list of interconnected hardware associated with a topology. Regardless of whether there is one type or multiple types of AI accelerators in a rack, the island is the unit of provisioning hardware resources, including AI compute.

Each AI Accelerator is associated with an ***AI Accelerator Hardware Type ID***. The context with the AI Acceleration Hardware Type ID designates features of the AI accelerator. It is possible for two different AI accelerators in the same island to support the same operator. The information associated with the ***AI accelerator island type ID*** will indicate the operators which can be executed on different hardware.

Island software support

Each AI accelerator island provides the software infrastructure for a program to queue work for a kernel that supports a set of operators. Each kernel has some set of bits which the PolyArch operators are compiled to. The program supplies the kernel and when the program allocates a kernel from the island, the island is responsible for “compiling” the kernel into targeted hardware instructions which will be used to execute the kernel. Based on the kernel, the island compiler is free to decompose the work further, perhaps sharding the operation in smaller, in-parallel units of work or for decomposing a large matrix operation the same operation on smaller tensors.

There are also the decisions of the local island schedule that are opaque to the program. Due to the asynchronous nature, units of work will be queued to the kernels in an island and the island schedule will dequeue that work and initiate the operation – or, conceivably – a batch of operations.

We expect that island compilation infrastructure supports both static computation and some form of just-in-time (JIT) compilation to react to changes in the system configuration. For example, if in a heterogeneous AI accelerator island where two or more different AI accelerators types within the island are able to implement one operator. Kernel JIT-ing allows the island scheduler to provision another accelerator type to execute a kernel or operator in the kernel. The island compiler compiles the kernel operators to target the new AI accelerator and work in the kernel’s work queue is routed to the newly compiled program on the new AI accelerator.

Adding new AI accelerator to a PolyArch system

The PolyArch system will also provide management software tools for the specification of different island types.

If a vendor would like to add its hardware into a system, it needs to perform

1. Associate the hardware with a previously assigned ***AI Hardware type ID***
2. Associating the AI hardware accelerator type with a specific island type and installing the new accelerators in the hardware systems.
3. Publish the capacity (or a set of operators) the accelerator hardware can support as part of the information for an island type id.
4. Extend the island compiler to support the compilation of PolyArch operators to the native unit of execution for the new AI accelerator.
5. Add any scheduling rules (e.g., is gang-scheduling allowed) for the new accelerator hardware which the island scheduler should be aware of.

Reconfigurability of existing AI accelerator hardware

For AI accelerator hardware which can be reconfigured, the island scheduler may choose to reconfigure the compute or infrastructure to reduce power or increase utilization.

For example, an accelerator hardware could support a 128x128 matrix operation, but also supports the ability to be reconfigured to handle four concurrent 64x64 matrix operations. If the island scheduler sees many 64x64 matrix operations than 128x128 matrix operations, the scheduler could choose to reconfigure the hardware to 64x64 matrix mode to increase utilization and, if possible, recompile the kernel associated with the 128x128 matrix operation to utilize 64x64 matrix operations.

Another case of reconfiguration may occur due to AI compute resource failures at runtime. For example, if the interconnect between two compute accelerators (physical or logical) is broken, the resource manager is notified and may trigger a reconfiguration of resources to form a new topology with modified capacity and capability. Reconfiguration may further trickle down the application's resource DAG to kick off JIT compilation as proper.

4. Things not covered/Questions

Here are a list of things not discussed in the above document which need to be specified:

- Storage Island namespace scoping Rules (see Google Pathways section 4.6)
- Management components for tracking different instances of various ID types
- Program and Island monitoring and trace facility.
- Hardware management (e.g., BMC) will track utilization, power, temperature, etc: AI Accelerator island schedulers may want to utilize this information or know when the BMC makes a global decision about power use and impact on frequency, bandwidth, etc.

- Detection and handling of sparse data.
- Operator fusion
- Multi-tenancy: security, noisy neighbor, etc.
- Operator language hinting rules for a program to specify/request a kernel use a specific type of AI accelerator (e.g., explicit lowering)
- QoS or other hints for a program to annotate on kernels for the island scheduler to take into account.
- Do programs share island ids or are islands units of provisioning (which may be overprovisioned given the actual hardware backing the island).
- Accounting for communication languages, e.g., NCCL, MSCCL, etc.
- hw/sw versioning, backwards compatibility support and rules with the AI Hardware Type ID.
- RAS (Reliability, Availability and Serviceability) capabilities
- How do we extend the PolyArch operator language?

4.1. Checkpointing

This is a deep topic which requires a lot of thought. A similar problem is “cloning” - taking a copy of an active neural network, either to move it or make a modified version, which is useful in the space of digital-twinning.

Checkpointing can be used for a number of reasons which have different requirements: de-prioritization of work, moving compute from one island to another island for management, or providing a point in time to restart a computation in the event of some sort of failure. Each of these use cases have different requirements and will impact the global scheduler, local island scheduler, and protocols to restart.

Checkpoints could conceivably only be available on certain hardware islands due to hardware capabilities. It also depends on the granularity of restart, e.g., if one set of kernels fails or is canceled because they are taking too long, is it possible to just restart the affected portion of work?

- Activation checkpoints
    - Hierarchical save/restore through composed memory/storage tiers (refer to Section 4.3)
- State Checkpoint/Restore
    - Model state
    - Optimization states
    - Accelerator states and interconnect topologies

- Size, intervals

Appendix E. Topics for future discussions

1. Whole-stack view

To tackle the most challenging problems in computer science, a comprehensive whole-stack approach is typically essential. For instance, the introduction of the CHERI hybrid capability processor was prompted by the need to address an application-level security issue without an existing hardware solution. The lesson drawn from the CHERI project emphasized the importance of a holistic perspective, considering CPU design, operating systems, programming languages, applications, and formal methods. This necessitates collaborative efforts that break down traditional silos within computer science research.

Applying this approach to crafting the blueprint for an AI Polymorphic architecture is crucial. When defining use cases that existing processor architectures cannot adequately address, it becomes imperative to thoroughly document these requirements. This documentation then serves as a guide for processor vendors, fostering collaboration to develop solutions that span the entire computing stack.

https://www.lightbluetouchpaper.org/2012/10/16/acm-queue-interview-on-research-into-the-hardware-software-interface/

Quantum Computing algorithms before mature Quantum Computing stack

The maturation of the Quantum computing stack for widespread use may require several more years. Nevertheless, quantum computing algorithms have already found application in various use cases, often collectively termed Quantum AI. Given the proficiency of quantum algorithms in handling dense mathematical computations and high-bandwidth memory, they align well with GPUs. Consequently, it is imperative to incorporate quantum use cases into AI polymorphic architecture considerations at present, rather than postponing it to the future.

https://www.wsj.com/articles/gpus-transformed-ai-now-theyre-here-for-quantum-b4ecbeeb

2. Markup language for Polymorphic Architecture data definitions

OCP ODSA group has developed the Chip Data Exchange Markup Language (CDXML) specification [31], which was adopted by the JEDEC Solid State Technology Association. This common language mainly targets the exchanging standards of hardware physical properties (electrical, mechanical and thermal) between chiplets.

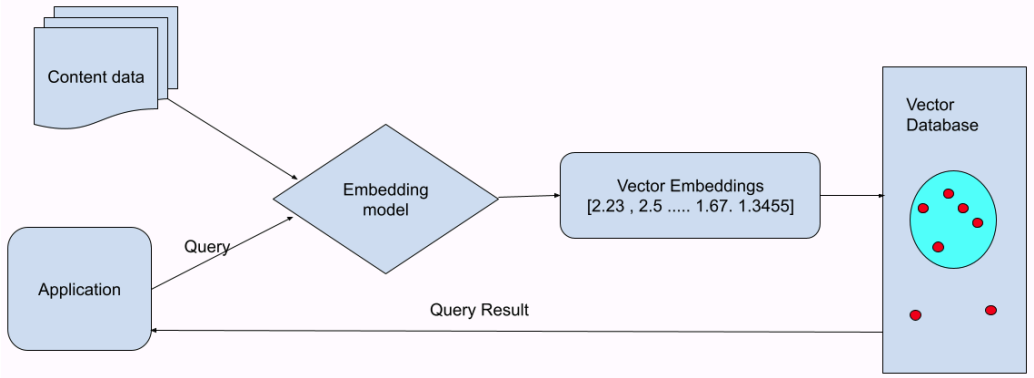
The polymorphic architecture system configuration, operation categories, and basic data formats may leverage (or adapt) the similar ideas to redefine the descriptions as outlined in Section 3.1. This future work may be a joint effort inside OCP.

3. Questions for on-going efforts inside OCP

Vector Database - Data Transfer is the bottleneck:

A Vector is a tensor with an 1-dimensional array of floating point numbers, generally a numerical representation of text , image or video which can be used to perform mathematical operations for similarity search.

An embedding is a vector of floating point numbers. A small distance between two vectors suggests that the two vectors are closely related. A vector database is a database that stores vector embeddings for similarity search and fast retrieval.



The separation of CPU and GPU is a main factor preventing vector databases from reaching full potential. While both CPU and GPU contribute to the overall computational power of a system, they are optimized for different types of tasks. CPUs are versatile and handle general-purpose computing tasks effectively, whereas GPUs excel in parallel processing and are particularly adept at handling the repetitive computations involved in vector operations.

Efficient data transfer between the CPU and GPU, as well as ensuring that each processor is utilized optimally for its strengths, becomes critical. This requires innovations in hardware architecture and software design to create a cohesive and well-integrated system that leverages the strengths of both CPU and GPU, avoiding bottlenecks and ensuring efficient vector processing.

Learn from the disruptive Innovation of Webassembly:

It is extremely hard to break out of existing ruts that the whole software ecosystem/industry trend to get stuck in. There is always huge value to stability and rich ecosystems of working code and "incrementalism hacks" like asm.js and webassembly are useful techniques to make big changes.

Webassembly is a very successful disruptive innovation that breaks down the existing abstraction layers. The team at Mozilla chose to focus on one of the oldest and hardest obstacles to progress: the performance of the JavaScript programming language. It started with Asm.js, a subset of JavaScript designed to be a low-level, performance-optimized language which allows developers to write code that closely resembles machine code, enabling more efficient execution in the browser.

WebAssembly introduced a binary instruction format that can be executed at near-native speed in the browser. Unlike JavaScript, WebAssembly is not human-readable and is designed as a low-level assembly-like language that can be targeted by compilers from various programming languages. This allows developers to write code in languages other than JavaScript, opening the door to a more diverse set of programming languages for web development.

Asm.js, then WebAssembly disrupted traditional abstractions by providing a more direct path to hardware-level execution. They allow developers to write code that is closer to the metal, reducing the overhead associated with higher-level abstractions. This can lead to significant performance improvements, making web applications more powerful and versatile.

In addition, these technologies break down the abstraction layers between different programming languages, enabling developers to choose the language that best fits their needs

and expertise. This flexibility is a departure from the traditional dominance of JavaScript in web development.

Various approaches can be taken to improve performance at the hardware level, future machines will likely use a variety of these techniques to meet performance goals.

4. Analog AI

A particular case of refactoring is the use of analog AI computing chips. The sigmoid activation function mentioned above is extremely expensive in regular math, and one would probably try to do it as a lookup table (+ interpolation), but it is quite cheap to perform in analog circuits since the equation fo $I = I_0 \left(e^{\frac{qV}{kT}} - 1 \right)$ r a diode is:

The main problem with analog is that it is difficult to communicate analog values at high speed between ICs, so the most likely approach is to die-stack an analog Non-Volatile Memory (NVM) technology with an analog computing base, which is a similar structure to Flash memory ICs. Essentially, Flash technology provides hundreds of layers of programmable resistors, but programming is slow, so it may be preferable to train models of such chips rather than the chips themselves. Given the variability in analog circuits, this may require digitally twinning actual chips for training, or doing in-place fine-tuning at deployment.

As well as lower power, the analog circuits may be more dense than digital circuits, offering higher capacity in the same space as digital chips.

Optical communication may be able to reliably communicate analog levels through phase in coherent laser systems, i.e. a reference signal multiplied by delayed version gives you a way to create amplitude modulation. Otherwise, single values can be transmitted using pulse-width modulation. Irregular short pulses of optical communication may be the best way to communicate low accuracy floating point values, since multiplexing and demultiplexing in optics does not affect the pulse characteristics (when splitting on color), and power is only used when a pulse occurs (a neuron firing).

5. Spiking Neural Networks

The matrix math approach to AI is generally performed by repeatedly cycling through data and applying a simple function to row & column elements (like multiply & accumulate). If the matrices

in question are sparse, one can unroll the operations into an event-driven structure, where any individual input change (a neuron firing) just ripples through and gives you a new result with minimum disturbance - most of the computational hardware being “dark” most of the time (not consuming power). This may require more hardware, but can be done in ways that avoid the power cost of continually cycling through the row/column calculations, which allows for denser packing of the AI. It also opens up opportunities to fine-tune per-neuron, one can assume that neural networks (like digital circuits) have critical paths.

6. Optical Computing

As mentioned above, multiplexing and demultiplexing different colors on and off a fiber connection can be done without incurring the delays seen in Copper (Ethernet), and amplitude can be modulated as well by splitting the path and modulating delays on the sub paths. That creates the opportunity to create “neurons” that sit between other computing blocks that act on signals between them. For example, multiple AIs may be trying to recognize things in a scene but the interest is only in a subclass, so the signals from the uninteresting subclasses can be suppressed in the optical network without incurring delays for the things of interest - maybe road signs for speed limits vs road signs for directions.

7. Quantum Computing

Most interesting AI work deals with large datasets and millions of parameters; quantum computing (QC) deals with small sets of parameters and polynomial equations - which it can solve instantaneously, although resetting the hardware may not be fast. QC fits into the bigger picture where you need a fast response to particular (rare) conditions, i.e. in a large neural network you would maybe try to engineer the critical path to use QC for particularly complex neurons (rather than the entire net). It would be combined with spiking neural networks to avoid unnecessary evaluation.