1. Introduction

The MemryX tower architecture is a **streaming**, **many-core**, **near-memory dataflow** design created from the ground up to accelerate AI workloads such as neural networks. In this document, we highlight the key concepts behind this architectural approach, explaining why each feature was introduced and how it impacts overall performance and efficiency.

We forgo traditional control-flow style architectures and elect to use a **Dataflow** compute paradigm, which natively aligns with the compute graphs that define Neural Networks (See Figure 1). In control-flow style architectures, a considerable amount of time and hardware resources are devoted to decoding instructions, computing addresses, and pre-fetching data. Data movement is inherently energy inefficient and transistor area is better spent on compute and storage than on-chip control-flow. Additionally, data routing and workload scheduling must be carefully managed to achieve any meaningful utilization of the hardware resulting in more complex software stack that often needs to be optimized on a per-model basis.

In contrast, MemryX Dataflow Architecture features powerful tensor compute-cores that are configured once at compile-time and then communicate directly with each other simply through the input and output data they consume and produce. Complex operators such as *Convolution* and *Dense* are supported at *the hardware level* with custom state machines, eliminating the need to decompose these operators into multiple instructions. Moreover, cores are able to each pass data directly to subsequent compute nodes without requiring a router or global scheduler. This enables a high level of scalability and design efficiency while streamlining data movement and execution.

MemryX Tower architecture consists of numerous heterogeneous dataflow cores—called MemryX Compute Engines **(MCEs)**—operating independently in a fully data-driven manner. Control is decentralized, enabling each MCE to process data as soon as it becomes available. The MX3 contains two main types MCEs: MAC cores (**M-Cores**) and ALU cores (**A-Cores**). By dividing the workload among multiple cores, the MemryX architecture naturally supports **space multiplexing**, allowing each neural network layer to be mapped efficiently and streamed through the dataflow

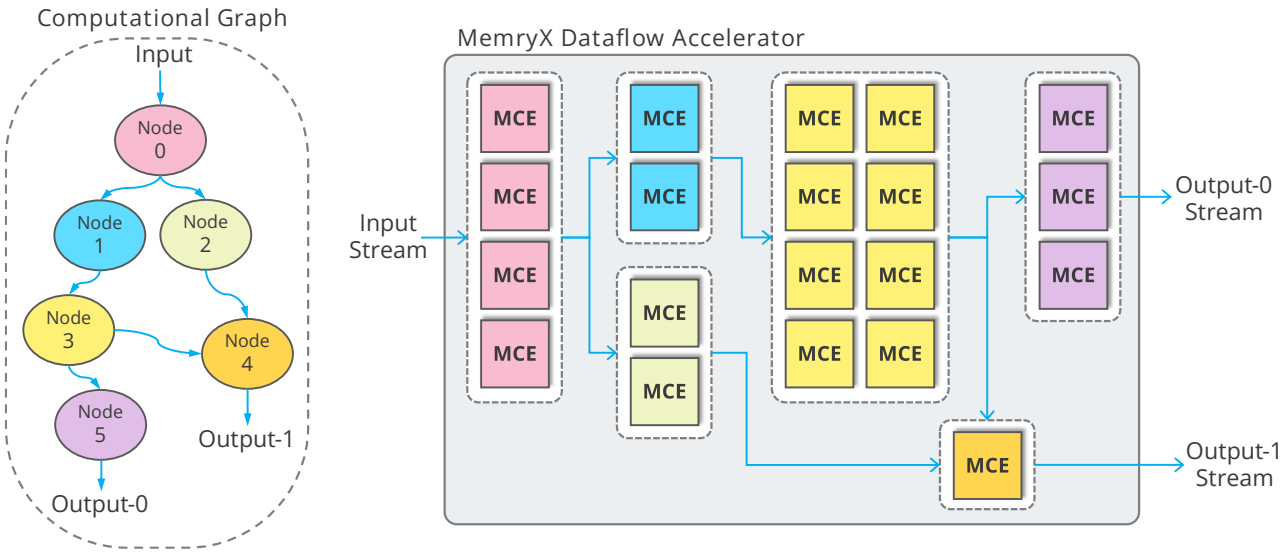

Figure 1 *- Illustration of the dataflow concept used in MemryX tower architecture.*

pipeline. MemryX modular dataflow architecture enables different types of MCEs to be implemented in the future chips while maintaining necessary compatibility.

On-chip distributed memories play a central role. Two key and separate memory types are used: **weight memories** that store neural network parameters, and **feature-map memories** that hold input, output, and intermediate data during inference. The feature-map memory also acts as the communication medium between processing elements, eliminating the need for a centralized on-chip memory and enabling software-defined data pathways. Each core or cluster of cores writes its partial results to the feature-map memory, allowing consumer clusters to read from this same shared buffer. This seamless data exchange reduces complexity by avoiding direct core-to-core communication and offers robust scaling options for larger systems and more complex models.

Cores and memories can be arranged in interleaving stacks that give rise to the "tower" nomenclature, as shown in Figure 2. Each compute tower contains an optimized number of groups of M-Cores and A-Cores alongside local weight memory, interleaved with feature-map towers. Every M-Core interfaces with local weight memory, adjacent feature-map towers, and neighboring M-Cores. A-Cores similarly interface with local LUTs (look up tables) and adjacent feature-map towers. This arrangement is optimized for efficient dataflow execution, as neural network inputs stream through multiple layers until reaching the final output.

The MemryX compiler programs the architecture offline and leverages a streaming execution model. Cores or clusters of cores are assigned specific tasks, processing incoming data and passing results forward in a pipeline. The compiler predetermines both how each core is programmed and how data flows between cores, ensuring balanced workload distribution and data coherence. One key objective is to maintain or create close producer-consumer relationships, strategically placing related operations near each other to maximize efficiency and minimize data movement.

Throughout the rest of this document, we will highlight how the architecture's features come together to enable efficient, scalable, and accurate acceleration of neural networks on MemryX hardware.

2. Hardware-Software Codesign

The MemryX tower architecture is designed from the ground up to be an efficient dataflow platform for AI workload acceleration. A key design priority is systematically balancing hardware and software elements, to achieve optimal utilization and seamless

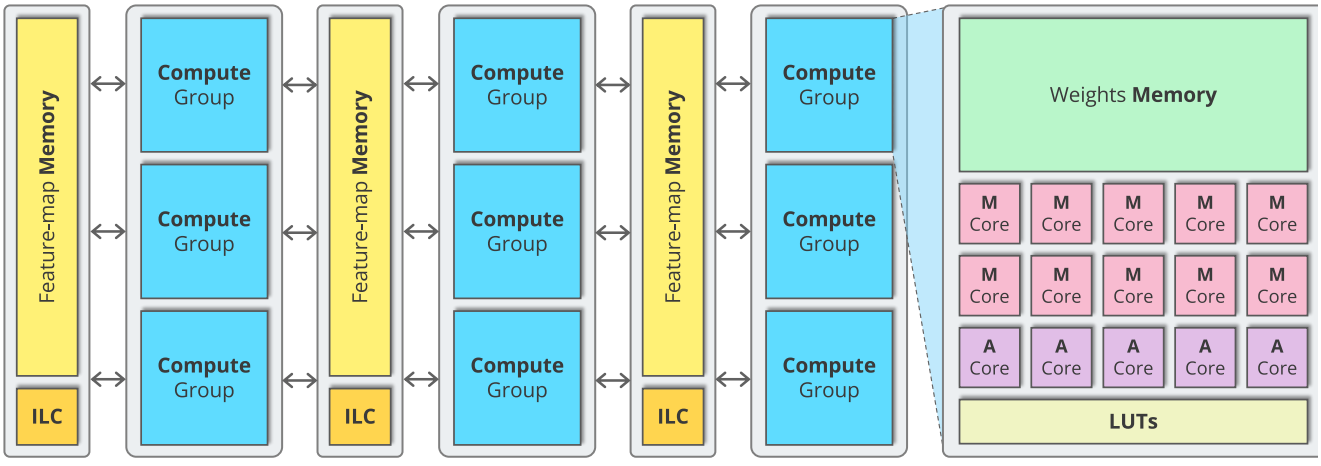

Figure 2 *- Illustration of the tower architecture, showing the interleaving stacks of compute and memory towers.*

usability. This co-design approach prioritizes efficient neural network acceleration while ensuring ease of use for developers. Through a flexible division of architectural features between hardware and software layers, the system attains peak performance and flexibility.

The steps to achieve an efficient co-design are as follows:

- **Step 1: Architectural Concept and Mathematical Modeling:** In the first phase, foundational architectural concepts are introduced based on the overall design goals, lessons learned from previous designs, and market needs. These concepts are then encapsulated in a comprehensive mathematical model containing hundreds of equations. Such equations capture both primary and lower-order properties of the architecture, providing a precise theoretical framework that guides subsequent design phases.

- **Step 2: Architecture Tool:** Next, the equations derived from the mathematical model are encoded into an analytical solver tool. This tool accepts various inputs—such as hardware constraints and the broad properties of target AI models, including layer counts and operators per layer—and solves them under these constraints to uncover an optimal design point. This stage often leads to the discovery and refinement of additional architectural features or design strategies, further optimizing performance and efficiency (see Section 4 for details).

- **Step 3: Detailed Modeling and Functional Compiler:** Once the core architectural data has been finalized, a detailed SystemC model is created. This model goes beyond the mathematical equations to incorporate higher-order considerations, such as memory arbitration and latency details. In parallel, a functional compiler is developed to target the SystemC model as its backend. This compiler operates with bit accuracy akin to the final hardware, while the timing is sufficiently estimated to be within ~10% of the chip's actual performance. Using this compiler and SystemC model, AI workloads are compiled and validated to ensure they meet the targeted levels of accuracy, performance, and utilization. Should these goals fall short, new hardware or software features are introduced and iteratively tested.

- **Step 4: Microarchitecture Design and Finalization:** By step 4, the architecture is stable enough for detailed microarchitecture design and front-end development. The SystemC model remains the default backend for the compiler until the chip is fully fabricated, acting as a reliable reference for ongoing development. As the compiler continues to evolve, it expands its feature set and performance optimizations while maintaining a solid hardware-software foundation. This holistic, iterative process ensures that the MemryX tower architecture delivers on its promise of efficient, user-friendly AI acceleration.

Throughout the remainder of this document, we will emphasize the principles of hardware/software co-design as we explore the architecture's key features.

3. Design for Scalability

A key aspect of our design philosophy is **modularity** and **autonomy** in the compute elements. By relying on a distributed-control, dataflow approach in both cores and memories, the MemryX architecture becomes **inherently scalable**. The size of the chip—along with its computational power—can now be **treated as a target-market** choice rather than a rigid architectural constraint, enabling efficient scaling to create the right

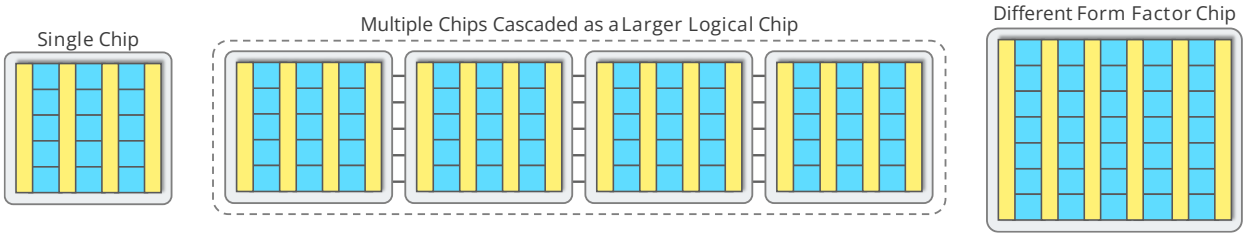

Figure 3 *- Illustration of MemryX scaling, from left to right: a single chip (left), multiple chips cascaded to form a larger logical device (center), and a physically bigger chip (right). This approach enables performance scaling both at the individual chip level and across interconnected modules.*

level of performance for a given application, from battery powered devices to data center applications. The MX3 design target was Edge AI applications such as industrial applications, security systems, Edge Servers, robotics, and more.

Moreover, this distributed-control concept extends beyond the chip level. Multiple MemryX chips can be **cascaded** within a single module, appearing to developers and users as a **single larger logical device**, as shown in Figure 3. Communication between these chips operates autonomously, with no host intervention required. The compiler automatically distributes workloads across the combined resources, treating them as a unified accelerator while each chip continues to function independently. Aside from ensuring data validity, there is virtually no synchronization overhead between chips, further enhancing both scalability and reliability in a wide range of deployment scenarios.

4. Design for Efficiency

Designing the architecture for efficiency means **doing more with less**. A solid design should extract the highest performance from the fewest resources. Accordingly, we strive to **minimize chip area**, **overall power**, **cost**, and **design complexity**, while simultaneously **maximizing performance** (frames per second), **hardware utilization**, **model coverage** (operator support), **target applications**, and **ease of use**.

Although the architecture is built on innovative principles (discussed throughout this document), these alone are insufficient for an **efficient design**. We must carefully determine how many compute units (MCEs) and memory blocks to instantiate, and how to organize and interconnect these resources. These architectural decisions profoundly influence performance and efficiency and must be made systematically.

Balancing Compute and Memory Bandwidth

Neural network inference primarily relies on vector/matrix multiply-and-accumulate (MAC) operations, implemented in our architecture via efficient fused multiply-adder blocks. Each MAC multiplies a feature map value by a weight value and adds the result to a running partial sum. To generate a single output value, numerous MAC operations are performed. A network layer computes many such values to form a complete feature map, and multiple layers work together to produce the final network output, resulting in millions or even billions of computations per inference.

While each output computation is generally independent, allowing parallel processing via multiple MAC units in a SIMD-like fashion, simply adding more compute can become wasteful if data (weights and feature-map values) cannot be supplied quickly enough from memory. Excess MAC units would remain idle, occupying chip area and power without boosting performance. Maintaining an optimal **balance between memory and compute** is crucial. Too many MAC units may cause the architecture to hit a **memory**

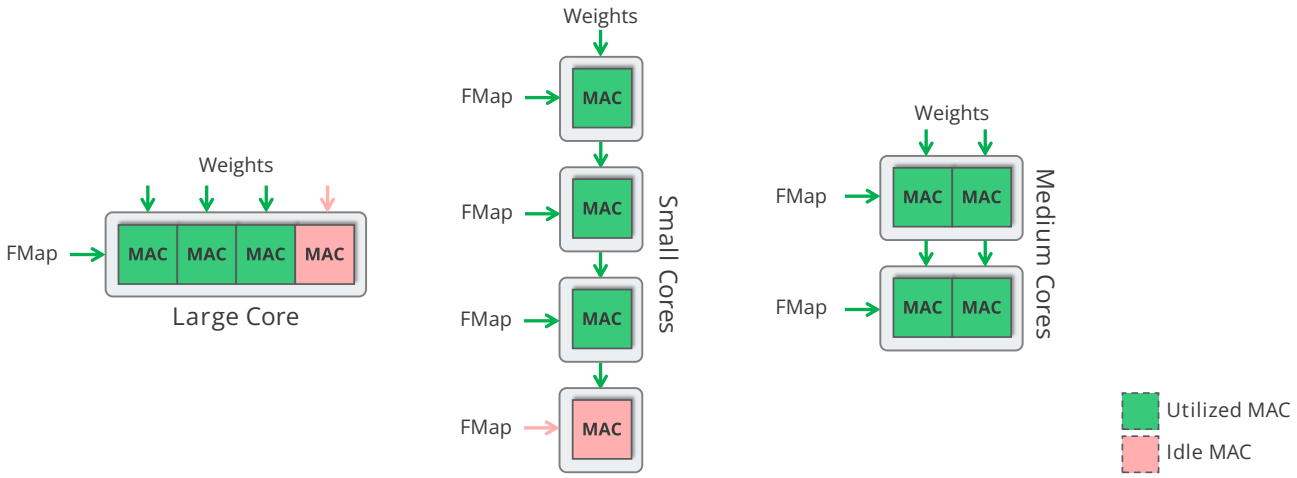

Figure 4 *- An example illustrating how adjusting core granularity influences feature-map and weight data bandwidth as well as data reuse—just one among hundreds of interconnected design parameters in the system.*

wall, where insufficient memory bandwidth leads to compute starvation. However, data reuse and other architectural techniques help mitigate bandwidth limitations by maximizing efficient usage of available memory resources, as shown in Figure 4.

The MemryX architecture balances memory bandwidth and compute to ensure high utilization. We achieve greater bandwidth by using on-chip distributed and segregated memories (see Section 5). High utilization combined with significant on-chip memory bandwidth enables MemryX architecture to outperform alternative architecture with much higher TOPS. In fact, the MX3 outperforms many AI computing systems with >10X more quoted peak TOPS.

Choosing an Optimal Design Point

We **systematically** determine the number and organization of MCEs, as well as the size, bandwidth, and hierarchy of on-chip memory. Balancing these elements is challenging due to the higher-order interdependencies of a dataflow-centric system. To address this, we encode our architectural innovations into a **large system of equations** representing the accelerator's behavior at a high level. We then constrain this system using **cost**, **performance**, and **power** targets, referencing a pool of representative neural network models to capture typical operations, memory needs, and operator support. With properly specified parameters, the system can be solved to yield an optimal or near-optimal balanced design point.

Once we arrive at this balanced design, we validate our choices using a co-designed architecture simulator and neural compiler, running real-world models to evaluate utilization and performance. Through iterative refinements—adjusting constraints and implementing improvements—we approach an optimal configuration (see Section 2 for details). The MemryX **scalable hierarchy** (MCE → MCE-Groups → Compute-Towers → MXAs) further organizes workloads across the chip. As a result, we can often deliver higher performance than competing solutions even with MemryX having far fewer raw compute elements, since our utilization is inherently more efficient.

5. Distributed Memory

Over the past few decades, the performance of processors has increased significantly, outpacing the improvements in memory performance. This growing disparity, often referred to as the "memory wall," has led to situations where the speed of computation is limited by the slower rate of data retrieval from memory. Traditional computer architectures use a hierarchical caching mechanism to exploit spatial and

temporal locality of program memory, aiming to hide the discrepancy between the speed of compute and the speed of data movement. However, the parameter counts of neural network workloads often exceed typical cache sizes, so locality becomes more difficult to exploit, and the throughput of the compute system becomes directly correlated with its memory bandwidth. Moreover, moving data from a distant source like DRAM consumes **an order of magnitude** more energy than the actual computation. Our architecture employs three main innovations to overcome these limitations.

First, we elect to use **on-chip memory**, thereby eliminating costly DRAM fetches and doing away with expensive off-chip interfaces. On-chip memory provides the necessary bandwidth, low latency, and minimal read power required to move relatively large parameter sets quickly and efficiently to the compute units. This also circumvents the overhead of memory caching hierarchies in which data must pass through multiple levels of memories before finally reaching the compute, resulting in many unnecessary reads/writes and wasted energy.

Second, we elect to **distribute** the memory across the chip, co-locating memory and compute units as opposed to having large caches separated from large compute clusters. By minimizing the distance between memory and compute, we greatly reduce the energy cost and routing complexities associated with the computation of a Neural Network. Incorporating distributed on-chip memory gives us the flexibility when designing the architecture to balance the compute throughput and memory bandwidth much more effectively than traditional architectures. This is because the number of memory blocks, their relative size, and the read-width / ports can be tuned to fit our needs.

Third, we separate the **two flavors of memory** that appear in neural networks:

- **Parameters** – Used to compute the inference results of a compute node (generally read-only).
- **Feature Maps** – Used to communicate inference results between compute nodes (frequently read and written).

Segregating these memory types allows us to leverage their unique properties. Parameter memory generally belongs to a single layer and is read-only, while feature-map memory is shared among multiple layers and experiences continuous reads and writes. We place parameter memory **within** MCE-Groups and feature-map memory **between** MCE-Groups. In doing so, weight memory can serve the cores within its group directly, whereas feature-map memory handles inter-layer communication. We further leverage memory specialization by optimizing the types of memory and their characteristics to better suite their role. For example, we can opt for high-density, read-optimized memory blocks for the parameter memory. In contrast, we employ mutli-port memories with equal read/write speeds for the feature map memories to serve as the Inter-Layer communication fabric (see Section 7 for details).

Finally, the unique memory principals that underline our architecture means we are positioned to leverage **new and emerging memory technologies** that align with these types of workloads. Breakthroughs in Neural Network optimized **RRAM or MRAM** technologies can be easily incorporated into our architecture leveraging the higher-density, read-oriented, and non-volatile nature of these technologies.

6. Dataflow Cores

The MCEs are the compute cores designed from the ground-up for the MemryX dataflow architecture. Their designs are focused on three key aspects:

1. No Instruction Fetch: In traditional control-flow architectures, each core must fetch both data and instructions. By contrast, the MCEs operate autonomously, relying on configuration registers and state machines instead of typical instructions. This approach eliminates instruction-fetch overhead and frees memory bandwidth for data transfers.

2. Asynchronous Submodules: Consistent with the broader MemryX architecture, MCEs are **data-driven** and feature **no centralized control**. Each core is divided into three independently-operating submodules or "stages":

- **Data Fetch:** The "front" of the core, it retrieves one feature map from on-chip memory and, if needed, weights from weight memory. It also manages Inter-Layer Communication (ILC) transactions. In a compute cluster, a core can also read data from nearby cores within the same Core Group.
- **Compute:** Fetched data is then passed to the Compute stage, which executes the assigned operation (e.g., Convolution, Addition). Meanwhile, the Fetch stage is free to continue work asynchronously, retrieving the next required data. When the Compute stage finishes its operations, it hands its results to the Writeback stage.
- **Writeback:** Receives processed feature-map data from the Compute stage, writes it back to feature-map memory, and completes any remaining ILC operations.

3. Deterministic Execution: The deterministic nature of the cores is vital for role within the overall dataflow architecture. The number of cycles each core takes to Fetch, Compute, and Writeback is carefully controlled in order to fall within expected memory read/write latencies. This deterministic design allows the Neural Compiler to accurately predict performance when mapping neural networks to the accelerator, and allocate resources accordingly.

Heterogeneous Cores

While a single, general-purpose core design could simplify software mapping, it risks having "idle silicon" if seldom-used features occupy chip area. To address this, MemryX employs a **minimally heterogeneous** design with specialized core types each focusing on different sets of operations that typically do not overlap (See Figure 5).

M-Core (MAC Core): The M-Core is a high-throughput vector/matrix compute core specialized for feature-map-by-weight operations, such as Convolution, Dense (Linear) layers, and striding window functions (e.g., Pooling, Upsampling). It can also handle common activation functions. Rather than using sequential instructions, the M-Core's ISA is parameter-based, relying on state machines that count to configured thresholds. This design allows the M-Core to dedicate most of its power and area to multiply-accumulate (MAC) units, maximizing throughput for core neural-network operations. A set of configuration registers defines the operation type, feature-map/kernel shapes, strides, pooling properties, and

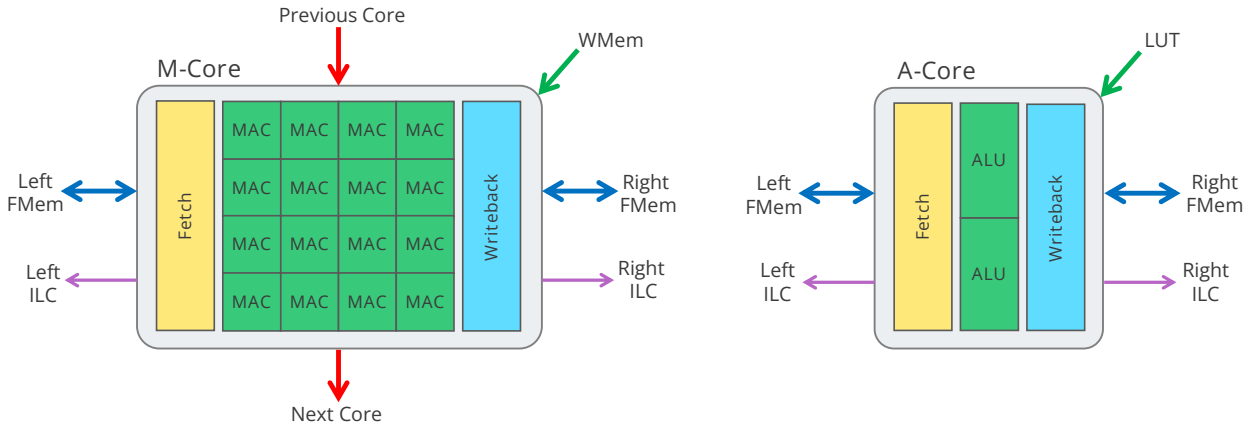

Figure 5 *- A top-level view of the M-Core and A-Core organizations, highlighting the three stages (Data Fetch, Compute, Writeback) and illustrating how each core type connects to memory resources and adjacent cores.*

memory addresses. Once set, the M-Core runs autonomously, counting up to the specified limits and taking appropriate actions when those counts are reached.

A-Core (ALU Core): The A-Core handles feature-map-by-feature-map operations (e.g., Add, Multiply, Concat) and specialized arithmetic (Reciprocals, Softmax, LUT-based approximations). Its microarchitecture shares similar Fetch and Writeback stages with the M-Core, but the Compute stage differs significantly. In Compute, the A-Core uses a RISC-like ISA with instructions and data registers, along with LookUp Tables (LUTs) for approximating complex functions. All instructions are stored in configuration registers within each A-Core instead of an external SRAM, eliminating instruction-fetch overhead and allowing flexible chaining of simple operations into more complex ones.

By pairing M-Cores with A-Cores, MemryX achieves a **balanced** hardware solution. This arrangement prevents unnecessary resource usage, avoids idle hardware, and maintains high efficiency across diverse neural network operators.

Graph Engine

Building on the strengths of the heterogeneous cores, the MemryX architecture natively supports a key set of **hardware-accelerated operators** (e.g., Convolution, Dense, Pooling, Add). Moreover, operator fusion techniques allow us to merge frequently adjacent operations (e.g., Convolution + BiasAdd + Activation), drastically reducing memory reads and writes. However, this hardware configurability must be carefully balanced against area, power, and complexity constraints. Therefore, MemryX leverages sophisticated graph processing to extend the range of supported operators through software. The **Graph Processing Layer** in the Neural Compiler, co-developed with the hardware, has deep insights into the capabilities of MCEs.

The graph engine iteratively applies various transformations to **optimize** the compute graph for the MemryX architecture. These transformations include **fusion** (combining adjacent operators into a single node), **conversion** (mapping unsupported operations to equivalent, hardware-friendly ones), **decomposition** (breaking complex operators into simpler sub-operations), **recomposition** (merging multiple sub-operations into a single high-level function), **reordering** (altering operation order for greater efficiency), and **approximation** (using simpler algorithms or LUT-based methods when direct hardware support is lacking). By applying these steps, the graph engine **maximizes performance** and extends operator support well beyond the raw hardware features—allowing a broad array of neural networks to run efficiently on MemryX accelerators while provide very high accuracy outputs.

While some of graph optimization steps, such as batch normalization fusion, are generally hardware independent, a majority of the graph processing steps are implemented to specifically **optimize** neural network **execution on the MXA**. Our custom Neural Compiler and MXA architecture were co-designed, so the hardware DNA is tightly encoded into our software stack, allowing us to achieve maximal utilization of the MXA hardware.

7. Through Memory Communication

A typical neural network model can be represented as a compute graph, with each layer's output (the **feature map**) serving as input to one or more subsequent layers. When this graph is mapped onto MemryX hardware (MXA), the workload is distributed across multiple MCEs (see Section 8 for details). Each core or compute cluster acts as a **consumer** of feature-map data originating a **producer** upstream core/cluster, while its own outputs can serve as inputs to another downstream cluster. This producer–consumer paradigm defines how data flows throughout the MemryX architecture.

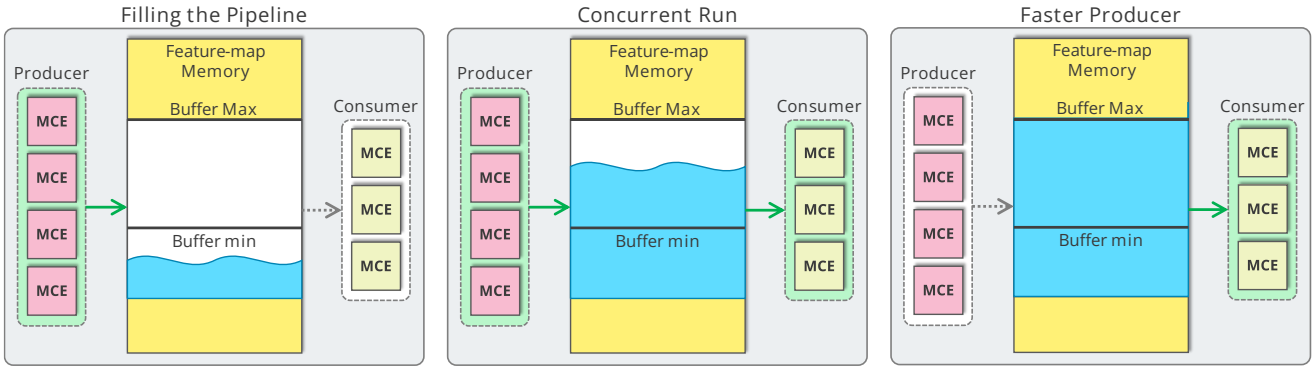

Figure 6 *- Illustration of the ILC reservoir concept in three modes of operation: (1) Pipeline fill (left): The consumer must wait for the producer to generate sufficient data. A similar scenario occurs in unplanned workload situations where the producer generates data at a slower rate than the consumer. (2) Optimal concurrency (middle): Both producer and consumer operate in parallel while data remains within the reservoir's minimum and maximum thresholds. (3) Faster producer (right): The producer outpaces the consumer. This can also occur due to backpressure from the consumer output.*

Reservoir Concept

To facilitate communication between clusters, MemryX employs a **shared** feature-map memory buffer, allowing the producer to write data and the consumer to read it directly—no router or on-chip network is required. A sophisticated **inter-layer-communication module (ILC)** manages **data synchronization**: for instance, a consumer cannot begin computing until the producer has generated enough data. The ILC enforces this by preventing data fetches in the consumer until the producer has finished writing.

In a less sophisticated design, a producer cluster would generate an entire feature map before any consumer could start. This approach is inefficient in both time and space, as it demands large buffers for intermediate feature maps and blocks the consumer from running while the producer is still active. By contrast, MemryX exploits neural network properties to enable **adjacent clusters** to work simultaneously, as illustrated in Figure 6. Once enough data is available, the ILC allows the consumer to begin processing while the producer continues operating on the rest of the data. Carefully structuring the order of computation discards data that is no longer needed, significantly reducing the memory footprint for intermediate feature maps.

Adjacency

While the **ILC reservoir** concept offers a straightforward and efficient way for MCEs to exchange data in a dataflow manner, it does rely on the assumption that both producer and consumer cores share access to the same feature-map tower. This requirement implies an adjacency between cores that need to communicate. However, in more complex neural networks or computational graphs direct adjacency is not always possible since far-apart layers may need to exchange data.

This is **where software-hardware splitting** comes into play (see Section 2). Instead of relying on a full on-chip network or router-based approach, **short-distance** (adjacent) communication is handled locally by the ILC reservoir as discussed. Meanwhile, **long-distance** communication is addressed at **compile time** through the mapper's place-and-route mechanisms. Whenever preserving direct adjacency is not feasible, the compiler can insert lightweight **bypass nodes** in the dataflow graph, as shown in Figure 7. These bypass nodes ensuring the adjacency requirement is met—even when layers must be placed far apart.

The compiler may also choose to **create** adjacency rather than merely **preserve** it if doing so results in a more optimal placement, routing, or performance

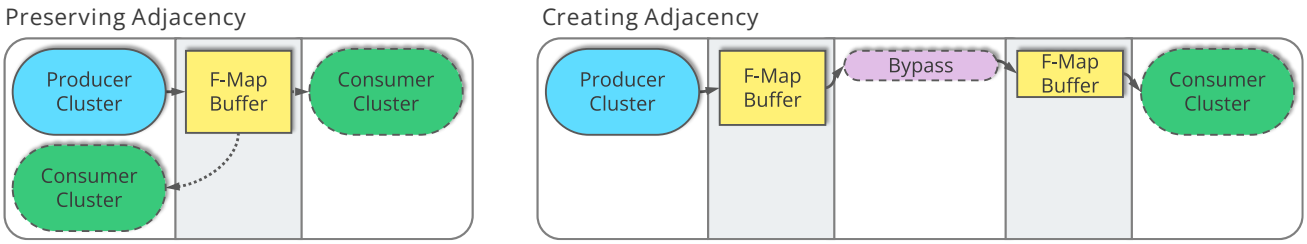

***Figure 7** - A diagram illustrating how the compiler can preserve adjacency by placing the consumer in the same or an adjacent feature-map tower, or create adjacency through a lightweight bypass node when layers must be placed farther apart.*

outcome. For example, the mapper might position a layer farther away in order to assign more compute cores, if overall performance benefits by inserting the additional bypass nodes.

Flow Simulator

The first two aspects of data movement—**ILC** and **adjacency**—lay the groundwork for transferring data across the tower architecture. Another key factor is the buffer size that must be allocated to each producer–consumer pair, along with the corresponding **ILC parameters**. Because on-chip feature-map memory is limited, these allocations must be carefully computed. For instance, one consumer might process multiple rows at once, while another handles just a single row; the producer might overwrite data only after it has been fully consumed.

The compiler applies a set of **equations** to determine each buffer's size and ILC settings, enabling **concurrent** rather than step-locked operation. More intricate structures, like **branch-and-merge** or nested loops (which are common in modern neural networks), can risk deadlock if a branch runs faster than the rest and lacks sufficient buffering. To address these issues, a **Flow Simulator** detects hazards—such as potential deadlocks—and assigns extra buffer resources or reorders tasks to maintain smooth pipeline performance. It also helps reduce stalls (bubbles), balance throughput across branches, and minimize memory access conflict.

In scenarios with large feature-map requirements, the compiler can revert certain producer–consumer interactions to **step-lock** mode. While this lowers peak performance, it ensures that high-memory workloads fit within available resources, striking a practical balance between efficiency and real-world constraints.

8. Workload Distribution

Workload distribution is pivotal to the MemryX tower architecture. By leveraging **hardware flexibility** and **advanced compiler** techniques, neural network layers can be split and mapped across multiple MCEs to maximize performance and resource utilization. Within a dataflow paradigm, every layer (or computational graph node) functions as both a producer and a consumer (see Section 7 for details).

At a higher level, the MemryX compiler transforms the original neural network (or multiple neural networks) into an **enhanced graph** (See Section 6 - Graph engine), where workload is assigned to a group of MCEs—known as a **compute cluster**. The compiler decides not only how many cores to allocate to each cluster but also how to distribute the workload among those cores, selecting from various techniques based on both the layer's properties (e.g., dimensions, channel count) and hardware specifications (e.g., weight memory bandwidth).

Distribution Strategies

1. Output-channel distribution: One fundamental approach is output-channel distribution, where each MCE computes a subset of the output channels across

all spatial dimensions (see Figure 8). Because cores do not directly exchange weight or intermediate data, synchronization overhead is minimal—only coarse-grained ILC producer/consumer checks are required. This strategy often excels for layers with many output channels.

2. Pixel workers: Another method is pixel workers, where each core processes part of the spatial dimensions for all output channels. Rather than every core fetching the same weights, only the topmost core in a chain retrieves them and passes the data along. Each subsequent core reuses these weights for its assigned pixels, significantly reducing weight-memory traffic (see Figure 8). While highly effective for layers with large spatial dimensions, it may be less suitable for operations that produce only a few output pixels.

3. Compound distribution: A third option is compound distribution, which merges channel-slicing and pixel distribution. The layer is split into sub-layers (or slices of output channels), each further divided into pixel-driven tasks for a group of cores. This method enables MemryX to handle deeper, more complex layers by sharing weight data where it's most beneficial, and can even scale across multiple MCE groups or chips. Additional techniques—such as input channel–based distribution—may also be used, depending on the network architecture.

Crucially, all these distribution strategies are orchestrated by the MemryX compiler, which uses an **equation-based** model and an **optimizer** that iteratively refines each layer's mapping. The compiler's objective is to sustain high utilization across MCEs, avoid memory bottlenecks, and reduce unnecessary data movement.

9. Hybrid Precision Compute

Throughout this document, we have emphasized the **performance** and **ease of use** of the MemryX architecture. Another vital consideration is **accuracy**. Many accelerators on the market adopt fully INT8 computations for speed, which can be fine for static parameters (weights) because they can be quantized offline. However, extending INT8 to **feature-map** data, which is inherently dynamic and depends on real-time inputs, introduces a significant challenge. Such accelerators often require a "tuning" phase with a representative dataset to establish scaling factors, and even then, accuracy can deteriorate if actual inputs differ from the tuning data. This process also complicates deployment, as collecting and tuning with real-world data becomes an extra step—and the results are only as reliable as the data used.

An alternative to INT8 is to store feature-map data in a floating-point format (e.g., BF16), preserving the network's dynamic range and accuracy. Unfortunately, BF16 effectively **doubles** the memory bandwidth and storage requirements compared to INT8, leading most solutions to offer either fast INT8 paths or slower

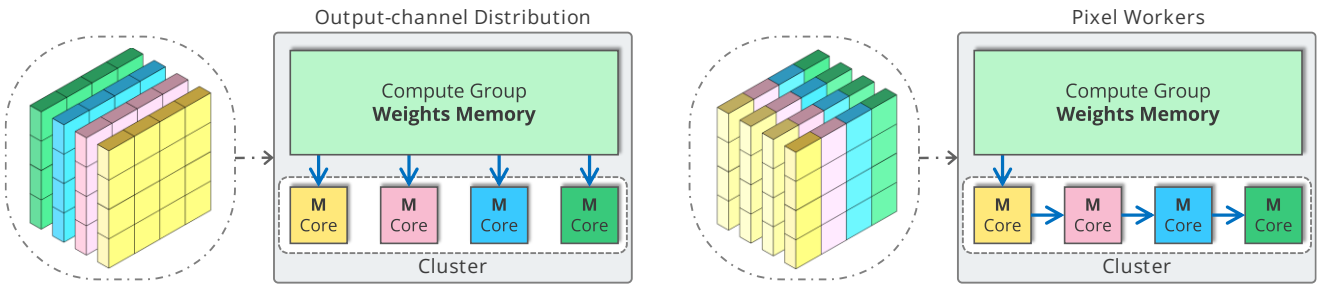

Figure 8 *- Illustration of two workload distribution strategies for a single neural network layer: output-channel distribution (left), where each core handles a subset of output channels, and pixel workers (right), where each core processes distinct spatial regions using shared weight data. Notably, in pixel-worker mode, only one core needs direct access to the weight memory, thereby preserving memory bandwidth.*

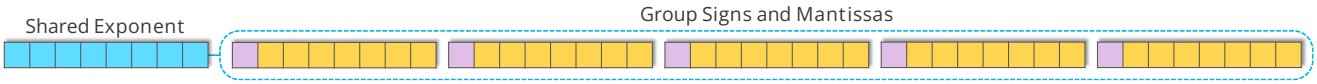

Figure 9 *- Group-BFloat format balances memory efficiency and accuracy by sharing an exponent across multiple mantissas, reducing bandwidth and storage overhead while maintaining dynamic range.*

floating-point options—or in many edge accelerators, only INT8 capabilities. Users are then forced to choose between sacrificing accuracy or accepting reduced throughput.

MemryX offers an optimized solution using **Group-BFloat**, a specialized format for feature-map data. Group-BFloat provides sufficient dynamic range, similar to BF16, but shares the exponent across multiple mantissas, as shown in Figure 9. This arrangement substantially reduces bandwidth and storage overhead while still accommodating the variability of neural network feature maps. Although sharing an exponent may not be ideal for every general-purpose use case, it aligns effectively with the statistical behavior of neural network activations. Furthermore, this is a parameter designed to be overridden should even higher precision be required for specific layers.

When **Group-BFloat** is used for feature-map data alongside **quantized weights**, the MemryX tower architecture delivers the **best of both worlds**. This combination not only achieves performance levels on par with an INT8/INT8 pipeline but also preserves accuracy comparable to traditional floating-point computations.

Efficient MAC

To realize these **hybrid precision** operations in hardware, each MemryX Core Engine (MCE) relies on multiply-accumulate (**MAC**) units at its computational core. A MAC unit accepts weight values, stored primarily in INT8 or INT4 (though INT16/INT32 are also supported via software layers) and input activations in Group-BFloat format. It multiplies these inputs and accumulates the resulting partial sums in an internal register. In many cases, the MAC sums tens of thousands of such products before producing its final output. By adjusting the sequence in which weights and activations are fed to the MAC, the same hardware can efficiently handle various vector/matrix operations.

Within the current generation, INT8 remains the default weight format, while INT4 halves memory usage (or doubles parameter capacity) but may introduce additional accuracy considerations. Meanwhile, Group-BFloat is the format used for input activations, sharing an exponent across multiple pixels to reduce bandwidth and storage requirements. Because the MAC sits at the heart of each MCE's compute capabilities, its design is **pipelined** for high throughput, delivering two floating-point operations (a multiply and an add) per clock cycle. By leveraging a shared exponent in the Group-BFloat format, certain logic can be **reused**, further reducing **power consumption** and **silicon area**.

10. Summary

The current generation of the MemryX tower architecture introduces **innovative dataflow principles** that enable a **highly efficient**, **easy-to-use edge** AI accelerator. By co-designing both hardware and software, MemryX achieves a balanced partition of features between silicon and the compiler/tooling layers. Through modular, distributed control and on-chip distributed memories, the system attains scalable performance with minimal off-chip data transfers and high resource utilization.

In this document, we followed a top-down narrative to explore the **high-level concepts** behind the MemryX tower architecture—from dataflow execution and hybrid precision strategies to heterogeneous cores and distributed memory. These insights provide a deeper understanding of how and why MemryX chips are designed, offering both developers and advanced users a clear view of the architectural decisions that enable MemryX products to excel.