BrainChip Akida as a Managed Resource in IBM Spectrum Symphony

Nikunj Kotecha
BrainChip Holdings Ltd
Laguna Hills, California, USA
nkotecha@brainchip.com

Ritik Shrivastava
BrainChip Holdings Ltd
Laguna Hills, California, USA
rshrivastava@brainchip.com

***Abstract*—BrainChip Akida is an event-based neural processor. It computes on the events a network actually produces, so the work it performs follows the activations present in a given input rather than the dimensions of the tensors that hold them, which makes it efficient at answering a single input quickly. IBM Spectrum Symphony dispatches very large numbers of short, independent tasks across a shared pool of machines, and has served workloads of that shape in production for two decades. The two point at the same operating regime, and this paper is a technical account of both platforms and of what joining them involves. On the Symphony side we describe its two-layer design, the objects an application is built from, the entitlement model by which a shared cluster is divided, and the mechanisms it already provides for managing resources that are not CPUs. On the Akida side we describe event-based convolution and the activation sparsity it turns directly into saved computation, the AKD1500 fabric, and the BrainChip SDK MetaTF toolchain that takes a trained model onto silicon. We then set out a reference deployment in which each compute node owns one AKD1500 device and presents it to the grid through Symphony's own metric reporting mechanism, and show how several devices can serve a single input larger than one device accepts. The implementation is published as a public repository that builds from public sources.**

***Index Terms*—neuromorphic computing; event-based processing; activation sparsity; grid computing; heterogeneous scheduling; edge AI**

I. Introduction

Inference, rather than training, is becoming the dominant consumer of artificial-intelligence compute. The International Energy Agency projects that data center electricity demand will more than double between 2024 and 2030, that the artificial-intelligence share of that demand will grow substantially over the same period, and that inference will account for the majority of it by the end of the decade [1]. That trajectory puts a premium on hardware that answers a single request cheaply.

Event-based neural processors are built for that case. They compute on the events a network produces, so a single input can be answered at low latency and low energy without first being accumulated into a batch [2].

The middleware that would place such work has had the least attention of any part of the picture. Surveys of neuromorphic hardware for the data center enumerate the available schedulers as Slurm, custom schedulers, and container platforms such as Kubernetes [2], and a recent evaluation of container orchestration for neuromorphic workloads examines a single lightweight Kubernetes distribution [3]. Commercial service-oriented grid managers have been dispatching this exact shape of work, very many small independent requests each wanted quickly, in production for two decades. They bring mature answers to the questions that arise as soon as a fleet of accelerators has to be shared: who is entitled to the hardware, how work reaches it, and what happens when part of it is unavailable.

IBM Spectrum Symphony is one such manager, and it has never been a CPU-only product. It documents graphics-processor scheduling at both the host and the device level [4], along with a harvesting add-on that admits idle accelerators as cluster capacity [5]. The question this paper takes up is what an event-based neural processor adds to that picture.

The contribution is a technical account. We explain Symphony in enough detail that a reader who has never operated a grid can follow the rest (Section 2), explain Akida with particular attention to event-based processing and activation sparsity, which is where its efficiency comes from (Section 3), and describe what joining the two involves (Section 4). Section 5 describes the implementation, which is published as a public repository.

II. IBM Spectrum Symphony

A. What it is for

IBM Spectrum Symphony is a workload management solution for applications made of many small, independent pieces of computation. The canonical example is a bank revaluing a portfolio: the calculation decomposes into hundreds of thousands of independent pricings, each running in milliseconds, and the useful answer exists only once all of them have returned. Work of this shape needs a workload manager whose cost of placing a piece of work is small relative to the work itself. An Amazon Web Services white paper on financial-services grids makes the point concretely: when a computation lasts one second, spending several more seconds deploying a container to run it is unacceptable [6]. Symphony's architecture is organized around driving that overhead down, principally by having worker processes already running and waiting before any work is submitted.

B. Two layers

Symphony is best understood as two layers with different jobs. Fig. 1 shows both, together with the Akida-equipped hosts beneath them that Section 4 describes.

The lower layer is the Enterprise Grid Orchestrator, usually written EGO. It is the resource layer, and it is concerned with what hardware exists, what condition it is in, and which parts of the organization are entitled to use it. On every host it runs a small set of agents that report status upward. Two are fixed: a load information manager that reports the standard metrics of a machine, and a process execution manager that starts and supervises things on behalf of the cluster. The third is not fixed, and is the interesting one. An external load information manager, universally abbreviated ELIM, is an operator-supplied program whose job is to report *additional* numeric metrics of the operator's own devising, on a schedule, into the same resource model the built-in metrics feed [7]. Anything an operator can measure and express as a number can become an input to placement decisions.

The upper layer is the service-oriented application manager, SOAM. It is where an application's own structure lives. A client submits work; a session director points it at the right place; a session manager takes ownership of a batch of related work and asks EGO for the capacity to run it; service instance managers start the actual worker processes on the hosts EGO granted; and the service instances themselves are long-lived processes that receive one piece of work at a time and return an answer.

C. The objects an application is built from

Four objects recur throughout this paper. An **application profile** is an XML document declaring everything Symphony needs to know about an application: what to run, where it may run, how many instances to pre-start, and how sessions should behave. A **service instance** is one running worker process pinned to one host; instances are started before work arrives rather than per request, which is how dispatch is kept cheap, since by the time a task is submitted a process is already waiting for it. A **session** is a batch of related work submitted by one client, with a lifetime, a type and a recovery policy. A **task** is one unit of work, an input payload in and an output payload out. Tasks within a session are independent, and Symphony's job is to spread them across the available service instances as fast as it can.

D. Sharing the cluster

Because Symphony grids are usually shared across an organization, entitlement is a first-class concern. Hosts are collected into **resource groups** by boolean expressions over their properties, so a group can be defined as "every host that is not a management host" or "every host with a particular characteristic". Organizational units are modeled as a tree of **consumers**, and a **resource plan** governs how capacity flows between them. Capacity may be *owned*, meaning a consumer is guaranteed a floor; *borrowed* from a consumer that is not currently using its entitlement; or *shared* from a common pool. The practical effect is that a specialized and scarce resource can be reserved for the consumers entitled to it without being stranded when they are idle.

Symphony is offered in several editions [8]. The work described in this paper uses **IBM Spectrum Symphony Community Edition 7.3.2**, which is available at no cost and provides the same functionality as the licensed editions within a 64-core ceiling. Where cluster sizes are mentioned below, they are Community Edition figures; the licensed editions scale to thousands of hosts and six figures of cores.

E. Resources that are not CPUs

Symphony's support for non-CPU resources is older and more concrete than is generally appreciated, and it establishes the pattern this paper follows.

For graphics processors, Symphony documents scheduling at two levels [4]. The *global* level decides which host a service instance runs on, using ordinary resource groups, which matters when only some hosts in a cluster carry accelerators. The *local* level decides which physical device on that host the instance binds to, at instance initialization. The separation is the right one, and it recurs below: placing a process on a machine and binding it to a device are different decisions with different constraints. A harvesting add-on extends the same idea to accelerators that are idle in machines owned by someone else [5], and a co-processor harvesting add-on has admitted accelerator cards as cluster resources, described as being treated much like compute nodes, since Platform Symphony V6.1 [9].

III. BrainChip Akida

A. Events rather than tensors

A conventional neural accelerator is organized around dense linear algebra. Activations arrive as tensors, are tiled, and are pushed through arrays of multiply-accumulate units that consume whatever occupies the tile. Throughput is a function of how many multiply-accumulate operations per second the array can sustain, and the design goal is to keep that array full.

Akida is organized differently [10]. Inputs are converted into discrete events, and computation is driven by the events that actually occur rather than by positions in a tensor that must all be visited. At the boundary of the chip a hardware conversion stage turns incoming data, pixels for instance, into the event representation the fabric consumes, so a conventional input can be presented to the device without a software preprocessing stage on the host [11].

B. Where the zeros come from

Trained neural networks are full of zeros, and not by design.

The rectified linear unit, the most common activation function in modern networks, maps every negative pre-activation to exactly zero. Whatever fraction of a layer's pre-activations happen to be negative therefore emerges as a hard zero, and that fraction is typically large: around half the activations in a representative network are zero, and training with an activity

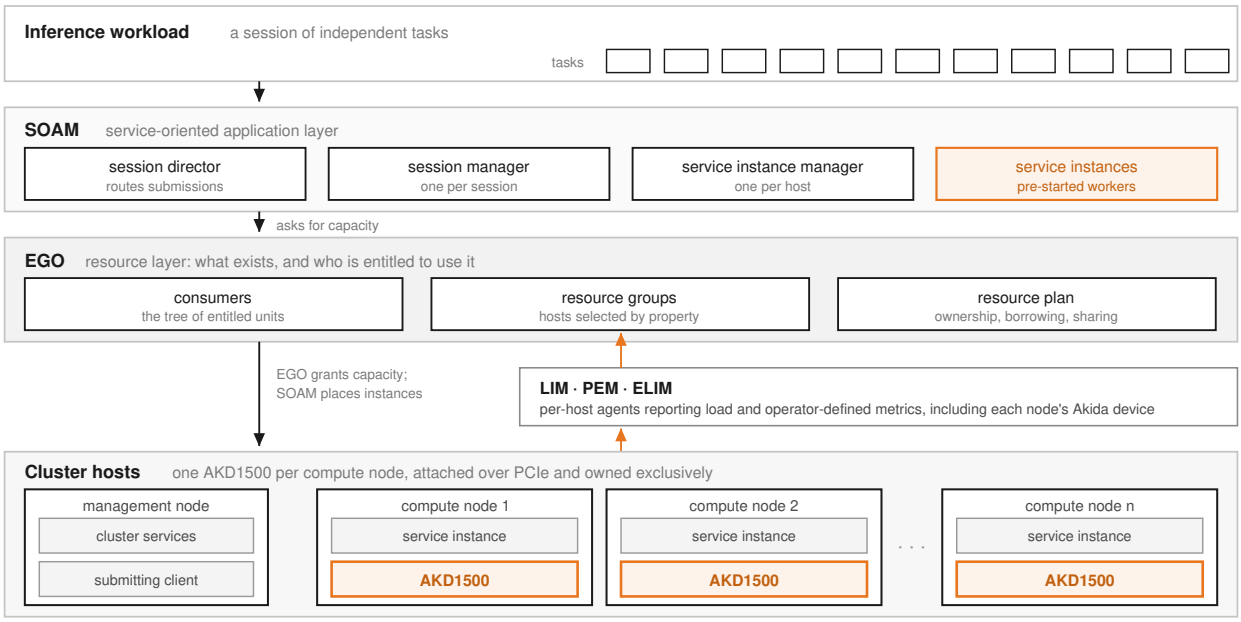

Fig. 1. The deployment described in this paper, read as layers. EGO decides what hardware exists and who is entitled to use it; SOAM turns an application's work into tasks and places them on the capacity EGO grants. Each compute node owns one AKD1500 device and reports it upward through the same per-host agents that report ordinary load, so the device becomes an input to placement decisions without any change to the layers above.

regularizer, which adds a penalty on activation magnitude to the loss, raises that figure past seventy per cent [12].

Two things are worth emphasizing. First, this sparsity is a property of ordinary trained networks: it is not something a model must be specially designed or retrained to exhibit. Second, the kind of sparsity matters. *Activation sparsity* is the proportion of zeros among the values a network produces as it runs, so it belongs to the data passing through the network and varies from one input to the next. *Weight sparsity* is the proportion of zeros among the trained parameters themselves, so it belongs to the model, is fixed once training is complete, and is what pruning sets out to create. The two call for different hardware, because skipping a zero-valued input and skipping a zero-valued weight are different operations. Akida exploits activation sparsity, and this paper makes claims only about that.

C. Event-based convolution

Everyone knows the zeros are there. The question is what it costs to avoid computing on them, and this is where the dataflow matters. Fig. 2 contrasts the two arrangements.

In a dense implementation the loop runs over output positions. Every position is visited and multiplied, and a zero operand costs the same cycle as any other. A sparsity-aware implementation on such hardware can do better, but only by first *finding* the zeros, which means testing operands, compressing them out, or maintaining index structures, and that detection is itself work performed on hardware whose efficiency depends on the dense array staying full. The saving is real, but it is a saving net of overhead.

Akida performs the convolution as an event-based operation. It iterates over the input values and projects each non-zero value's kernel contribution into the output space; where the input is zero, nothing is projected [12]. A zero is not detected and discarded. It is simply never enumerated, so no mechanism is required to skip it and no cycle is spent deciding to. The consequence is direct: the computation a layer performs follows the number of events it produces, which is the number of non-zero activations. Activation sparsity therefore helps to save computation, and hence the energy that computation would have cost, without a detection step in between.

D. The fabric

The processing resources are arranged hierarchically, as shown in the upper half of Fig. 3. Neural processing units are grouped into nodes, four to a node, and the nodes communicate over an on-chip mesh that carries both event traffic and configuration. Each unit has local SRAM for kernels and neuron state, and that memory can be reallocated between neighbors as a particular model requires.

The device used throughout this paper is the AKD1500 [10], summarized in Table I from its public product brief [13]. It implements the first-generation Akida architecture on a 22 nm fully-depleted silicon-on-insulator process and attaches to a host over PCIe. BrainChip announced it as commercially available and shipping in production volumes in June 2026 [14].

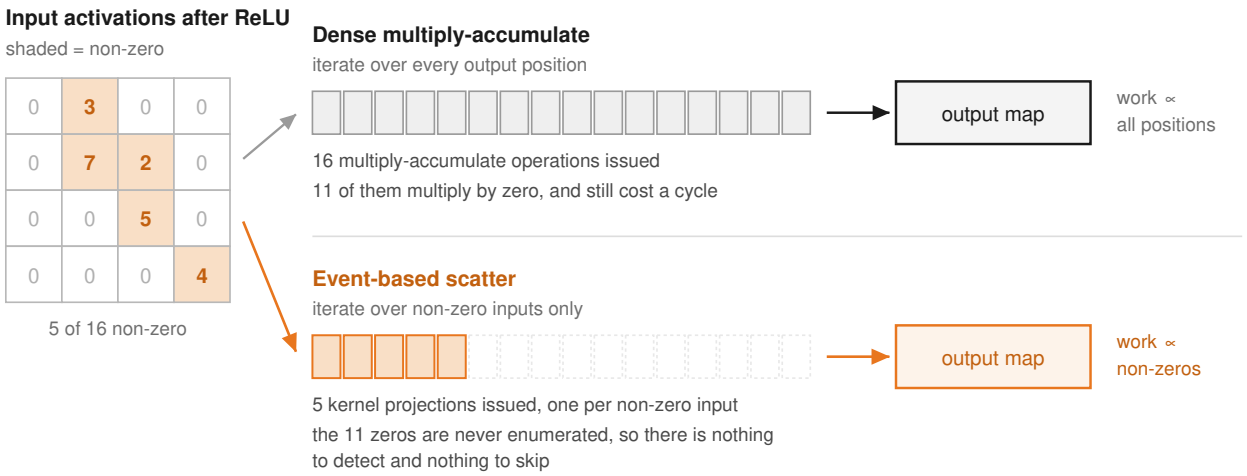

Fig. 2. The same convolution under two dataflows: a dense arrangement enumerates output positions and multiplies whatever occupies each one, so a zero still consumes a cycle; an event-based arrangement enumerates non-zero inputs and scatters each one's kernel contribution into the output, so zeros are never reached.

Table I
THE AKD1500 REFERENCE DEVICE.

Property	AKD1500
Architecture	Akida 1.0
Process	GF 22 nm FD-SOI
On-chip memory	1 MB
Clock	5 to 400 MHz
Host interface	PCIe Gen2 endpoint, or SPI

E. From a trained model to the device

Devices are programmed through BrainChip SDK MetaTF, a Python toolchain in three stages [15]. The lower half of Fig. 3 shows the path. A quantization stage reduces a trained Keras or PyTorch model to the low bit-widths the fabric supports; a conversion stage rewrites the quantized model into Akida's event-domain representation; and a runtime loads the result onto hardware.

The last step in that path is mapping, which places the converted model's layers onto the fabric. Where a model asks for more of the fabric than is configured at one time, the Akida Engine Library runs it in several passes and reconfigures the fabric between them, automatically and without involving the caller [16]. A caller that requires the whole model to be resident as a single sequence can ask for that explicitly and be told directly whether it was achieved, rather than inferring it afterwards [17]. Section 4.3 uses that facility as a readiness condition.

What a device accepts is documented in full [11]. Layer types, layer dimensions, kernel sizes, strides and pooling all have defined support, and the input conversion stage that turns image data into events accepts an input height between 5 and 256 pixels. Section 4.6 returns to that last figure, because a grid gives it a useful answer.

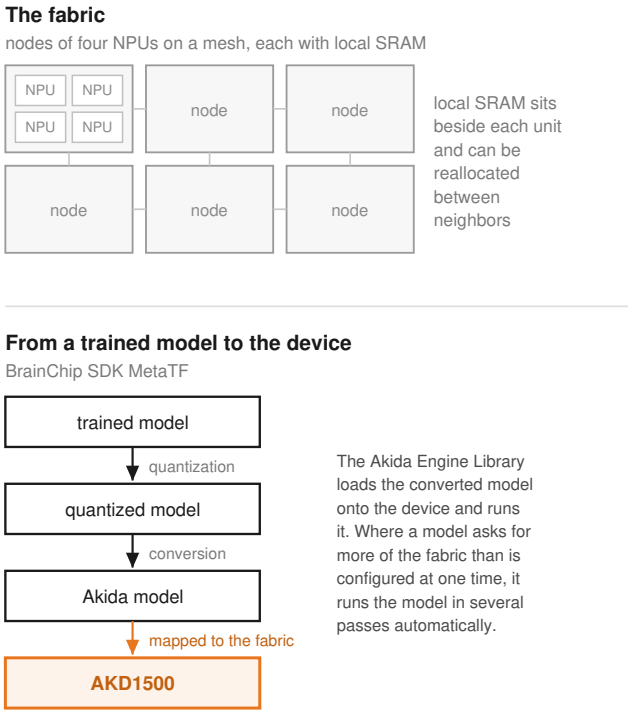

Fig. 3. Above: the processing fabric, with local memory beside the units that use it. Below: the path a trained model takes through BrainChip SDK MetaTF onto an AKD1500 device.

IV. AKIDA ON A SYMPHONY GRID

A. Why the two fit

Section 2 described a system built to dispatch enormous numbers of small independent tasks quickly, whose users

are organizations that already think of compute as a shared, entitlement-governed pool. Section 3 described a processor that is at its most efficient answering one input at a time.

These two descriptions meet. Symphony divides an application into many small requests and expects each one to be answered on its own, as quickly as possible. That is the regime in which an event-based processor is most efficient, because it is built to answer a single input rather than to fill a wide arithmetic array with many inputs at once. A SOAM task carries exactly one input, so the workload reaches the hardware in the form it is strongest on, with no batching stage in between.

The pairing also puts the right software in charge of the thing that determines what a fleet of accelerators actually delivers. Keeping devices busy, dividing them fairly between the consumers entitled to them, and reclaiming them when their owner has finished are exactly the problems a mature grid manager already solves, and they are the problems that decide how much of a device's efficiency an operator collects in practice.

B. What the workload manager needs to know

A grid can only place work on a resource it can see. For an Akida device that means three facts, none of them exotic: that the device is present on the host, that it is healthy and has been claimed successfully, and which model the service instance on that host is currently serving. Together these let Symphony distinguish a node that can take work now from one that cannot, and let an operator direct particular work to the nodes serving a particular model.

None of this requires modification to Symphony, because the mechanism for reporting it already exists. An ELIM reports operator-defined numeric metrics into the same resource model the built-in metrics feed [7], and resource groups select hosts by boolean expressions over exactly such properties. Device presence, readiness and the resident model are all expressible that way, which is what the upward path in Fig. 1 shows. This is also the integration point identified in prior work on the same combination [18].

C. A reference deployment

Fig. 4 shows the arrangement used in the implementation described in Section 5.

The topology is deliberately plain. One management node runs the cluster services and the submitting client. Each compute node is granted exclusive ownership of exactly one AKD1500 device and hosts one service instance bound to it. Symphony's resource groups keep the inference service off the management node and on the compute nodes, using the same boolean host-property expressions the product has always used.

Exclusive one-to-one ownership keeps the resource model as simple as it can be. One service instance, one device, nothing to arbitrate, and a node that either has its device or does not. It also gives readiness a clean definition. A node reports itself ready only after it has claimed its device and mapped its model, so Symphony places work only on nodes that can serve it, and a partially provisioned cluster runs correctly on the nodes that are healthy. Treating mapping as a readiness condition rather than as a runtime concern is what makes that behavior fall out of the design instead of having to be handled at dispatch time.

The cluster size in the reference deployment follows from the Community Edition core ceiling described in Section 2.4, which admits one management node and seven compute nodes on the hardware used. This is a licensing boundary rather than an architectural one.

Table II
TWO WAYS OF PUTTING WORK ON THE SAME FLEET.

	Serial round-robin	**Concurrent fan-out**
Transport	direct HTTP	a SOAM session
Dispatch	one at a time	spread by SOAM
Devices busy	about one	every ready node
Purpose	a baseline	intended setup

D. Two interpreters on one node

Symphony's Python binding and the Akida runtime target different interpreters. The binding is distributed as compiled artifacts built against the application binary interface of CPython 3.6; the Akida runtime is distributed as a modern Python wheel for CPython 3.10 or later. A single process therefore cannot both receive SOAM tasks and perform Akida inference, and the integration gives each runtime its own interpreter with a narrow, explicit interface between them. Fig. 5 shows the arrangement.

The service instance speaks SOAM. It supervises a long-lived worker process in the other interpreter, and that worker owns the device and holds the mapped model. Control messages, naming the model and the size of the payload, cross a pipe as small JSON headers. The tensor itself does not: it is written once into a shared memory segment and read in place by the worker, so the per-task cost on the node stays close to the cost of the inference and does not grow with the size of the input.

The lifecycle rule that makes this robust is the readiness condition from Section 4.3. The worker maps its model at startup and signals readiness only once the model is resident on the device. A service instance whose worker does not reach that state does not start, and Symphony's own health machinery does the rest.

E. Dispatch patterns

With the integration in place, the question becomes how work should reach the fleet. Table II contrasts the two arrangements the implementation provides.

The serial arrangement addresses each node directly over HTTP and issues one request at a time, cycling through the fleet. It is deliberately naive, and it exists to make the contrast visible: it is what a fleet of accelerators looks like without an orchestrator, and it keeps roughly one device busy at any moment regardless of how many are installed.

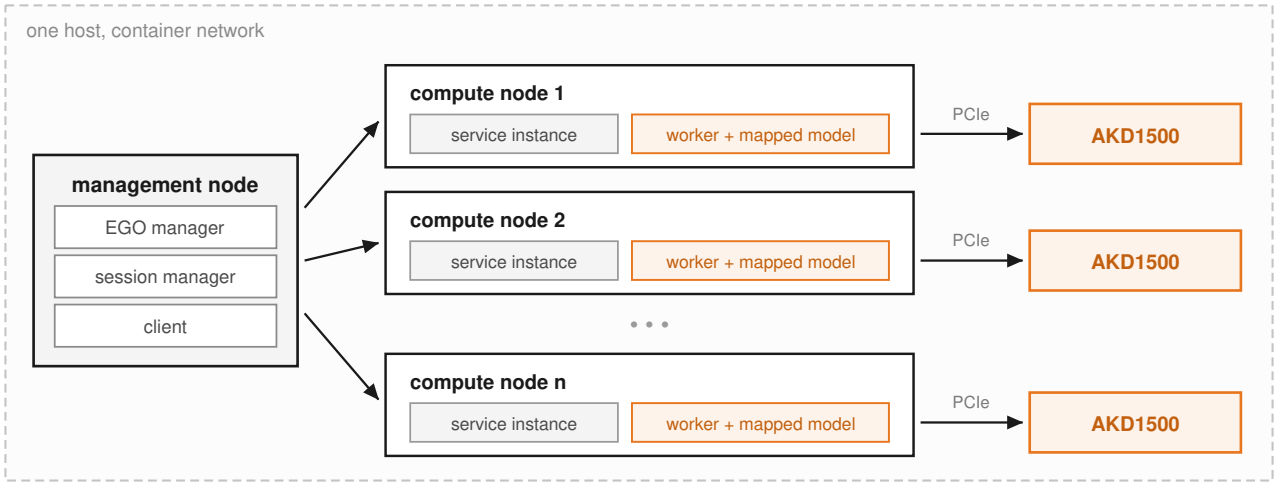

Fig. 4. A reference deployment. One management node runs the cluster and the submitting client; each compute node owns exactly one AKD1500 device over PCIe and hosts one service instance bound to it. A node reports itself ready only once it has claimed its device and mapped its model, so Symphony places work only on nodes that can serve it.

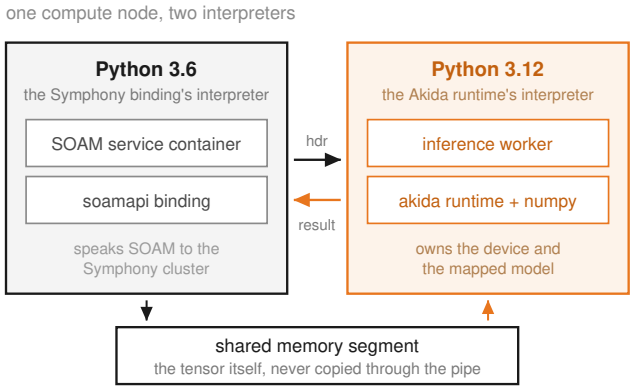

Fig. 5. Two interpreters on one compute node. The service instance speaks SOAM to the cluster; a long-lived worker owns the device and holds the mapped model. Control crosses a pipe as small headers, and the tensor crosses shared memory.

The concurrent arrangement submits the same work as tasks within a single SOAM session and lets the session manager spread them across every service instance. This is the intended configuration, and the useful observation is how little application code the difference requires. The worker is identical. What changes is that placement, concurrency, failure handling and retry become the middleware's responsibility rather than the application's.

F. Large inputs across several devices

An orchestrator does two useful things rather than one. It keeps many devices busy on independent work, and it lets several devices act as one for a single item of work. The input dimensions of Section 3.5 are where the second becomes concrete: the input conversion stage accepts an input height of up to 256 pixels, so an input larger than that is naturally expressed as a set of regions that each fit comfortably within it.

Because a grid's business is decomposing work and recomposing results, that expression is one the middleware already knows how to serve. An input too large for a single device is divided into regions, the regions are dispatched concurrently to different devices, and the results are merged afterwards. The implementation includes a three-stage SOAM pipeline that does exactly this for object detection on a 448 by 448 frame across six devices, where the substantial part of the work is the merge, since detections that straddle a boundary have to be reconciled rather than pooled naively; the details are documented with the source [19].

V. Implementation

The implementation described above is published as a public repository [19]. Two properties of it are worth stating.

It builds from public sources. A clone and a container build produce the running cluster, with the Symphony tree taken from IBM's own published container image and the Akida runtime from its public package index. No private registry and no vendor credentials are involved. Because the Symphony Community Edition tree carries IBM's own license terms, the build requires the operator to accept those terms explicitly and stops before contacting IBM's registry otherwise.

It verifies itself. The image ships an acceptance check that asserts the invariants the runtime arrangement depends on: that the Symphony binding imports under its own interpreter and the Akida runtime under its own, that the shared-library closure of the vendor binaries resolves, that environment resolution lands where it should, and that the credentials the cluster services need are valid. Each of these is cheap to check at build time and useful to have checked, which is the case

for asserting them explicitly rather than relying on a first run to reveal them.

VI. Outlook

Three directions follow naturally from the design.

The first is heterogeneous coexistence. Symphony already manages CPUs and graphics processors, and an event-based device fits alongside them rather than in place of them. A configuration in which an Akida device runs a continuous low-latency screening stage and a graphics processor is engaged only for the fraction of inputs that warrant more expensive analysis has been demonstrated as a two-stage pipeline on a single host [20]; expressing it as two consumer classes within one grid is a natural next step.

The second is the range of workloads this suits. Financial services are Symphony's home ground, where streaming classification, screening and risk pipelines all share the property that per-item latency matters more than aggregate throughput. Network security has the same character, since traffic and telemetry arrive as an unbounded stream of individually small decisions whose value falls quickly as they age. Drone fleets and other autonomous systems add a further constraint, because each vehicle has to decide locally within a firm power budget while remaining coordinated with the rest. Applying an event-based tier to work of any of these shapes is a direct extension of what the platform is used for today.

The third is scale. The reference deployment here is deliberately small, and the same software manages clusters several orders of magnitude larger, so nothing in the arrangement of Section 4 is tied to the size of the cluster it was built on.

Alongside these, characterizing the deployment quantitatively is work we intend to publish: dispatch overhead relative to on-device inference time, how throughput scales with fleet size, and how energy per unit of served work varies with utilization.

VII. Related work

The combination of Akida and Symphony is not new. Kevin D. Johnson of IBM has published a substantial body of work on it since late 2025, including a large corpus of numbered demonstrations [21] and technical articles on scheduling neuromorphic fleets with Symphony and LSF [18], on multi-tenant inference fabrics built over Symphony and a shared file system [22], and on orchestrating neuromorphic devices alongside quantum, graphics and mainframe resources in a single scheduling domain [23]. That work establishes the architectural feasibility of the pairing across a wide range of applications, and it identifies the ELIM mechanism as the integration point, a conclusion we reached independently and adopt in Section 4.2.

The wider literature on neuromorphic hardware in the data center approaches the question from the hardware side. The most complete survey of the area identifies hardware-software integration for production environments as one of three barriers to adoption and enumerates the available schedulers as Slurm, custom schedulers and container platforms [2]; a 2026 study of container orchestration for neuromorphic workloads evaluates a single Kubernetes distribution [3]. The same survey observes, usefully for our purposes, that PCIe-attached Akida boards represent a way of integrating neuromorphic computing into data centers much as graphics processors are integrated. Intel's Hala Point [24] and IBM's NorthPole [25] give context for neuromorphic systems at scale, and teaching an existing production scheduler about a new class of resource is a well-established pattern of which Slurm's generic resource mechanism is the canonical form [26].

VIII. Conclusion

Inference is becoming the dominant consumer of artificial-intelligence compute, and the demand it creates is made of single requests that each have to be answered quickly and cheaply. BrainChip Akida is built for that case. It computes on events, so the work it performs follows the activations a network actually produces, and activation sparsity, which ordinary trained networks exhibit without being designed for it, helps to save computation and hence the energy that computation would have cost, with no detection step to pay for first. Its fabric holds a network in memory beside the processing units that use it, and BrainChip SDK MetaTF takes a trained Keras or PyTorch model onto an AKD1500 device in three documented steps. The result is a processor that is efficient precisely where a single input has to be answered quickly.

That is the workload IBM Spectrum Symphony was built for. Symphony dispatches vast numbers of small, independent, latency-sensitive tasks across a shared pool, treats compute as an entitlement-governed resource that many teams draw on, and has managed silicon that is not a CPU for many years. A SOAM task carries a single input, which is the regime an event-based processor is strongest in, and the placement, sharing and recovery machinery a fleet of accelerators needs is already present and proven.

Joining them requires no modification to either product. A compute node claims one AKD1500 device, maps its model, and reports the device upward through the same mechanism Symphony has always used for facts about a host that it does not natively know. From there the grid does what it already does: it places work on the nodes that can serve it, keeps every ready device busy, and composes several devices when a single input calls for more than one. We have published the implementation, which builds from public sources, so that the arrangement can be examined and reproduced rather than only described.

Acknowledgements

We thank Kevin D. Johnson of IBM, whose demonstrations of Akida devices running under IBM Spectrum Symphony prompted this work and shaped our understanding of how the two products fit together.

of BrainChip Holdings Ltd. Other names may be trademarks of their respective owners. This paper describes an integration built on IBM Spectrum Symphony Community Edition.

References

[1] International Energy Agency, "Energy and AI." IEA report, Apr. 2025. Available: https://www.iea.org/reports/energy-and-ai

[2] B. Vogginger *et al.*, "Neuromorphic hardware for sustainable AI data centers," *arXiv preprint arXiv:2402.02521*, 2024, Available: https://arxiv.org/abs/2402.02521

[3] H. Pham and B. Silverajan, "Evaluating container orchestration for neuromorphic workloads in virtual edge environments," *arXiv preprint arXiv:2605.15866*, 2026, Available: https://arxiv.org/abs/2605.15866

[4] IBM Corporation, "GPU application configuration." IBM Spectrum Symphony 7.3.1 documentation, 2021. Available: https://www.ibm.com/docs/en/spectrum-symphony/7.3.1?topic=add-gpu-application-configuration

[5] IBM Corporation, "IBM spectrum symphony GPU harvesting add-on." IBM Spectrum Symphony 7.3.0 documentation, 2021. Available: https://www.ibm.com/docs/en/spectrum-symphony/7.3.0?topic=features-spectrum-symphony-gpu-harvesting-add

[6] Amazon Web Services, "Financial services grid computing on AWS: Task scheduling and infrastructure orchestration." AWS Whitepaper, 2020. Available: https://docs.aws.amazon.com/whitepapers/latest/financial-services-grid-computing/task-scheduling-and-infrastructure-orchestration.html

[7] IBM Corporation, "Elim – external load information manager." IBM Spectrum Symphony 7.3.1 reference, 2021. Available: https://www.ibm.com/docs/SSZUMP_7.3.1/reference_sym/elim.html

[8] IBM Corporation, "IBM spectrum symphony editions." IBM Spectrum Symphony 7.3.1 documentation, 2026. Available: https://www.ibm.com/docs/en/spectrum-symphony/7.3.1?topic=foundations-spectrum-symphony-editions

[9] IBM Corporation, "IBM spectrum symphony co-processor harvesting add-on." IBM Spectrum Symphony 7.3.1 documentation, 2021. Available: https://www.ibm.com/docs/en/spectrum-symphony/7.3.1?topic=add-spectrum-symphony-co-processor-harvesting

[10] BrainChip Inc., "Akida neural processors." BrainChip product pages, 2026. Available: https://brainchip.com/chips/

[11] BrainChip Inc., "Akida 1.0 hardware capabilities." MetaTF documentation, 2025. Available: https://doc.brainchipinc.com/user_guide/hardware/1.0.html

[12] D. McLelland and A. Kayyam, "Akida exploits sparsity for low power in neural networks." BrainChip technical article, 2025. Available: https://brainchip.com/akida-exploits-sparsity-for-low-power-in-neural-networks/

[13] BrainChip Inc., "AKD1500 edge AI co-processor product brief, V2.4." Oct. 2025. Available: https://brainchip.com/wp-content/uploads/2025/10/AKD1500-Product-Brief-V2.4-Oct.25.pdf

[14] BrainChip Inc., "BrainChip announces commercial availability and production shipments of AKD1500 neuromorphic processors." Press release, Jun. 2026. Available: https://brainchip.com/brainchip-announces-commercial-availability-and-production-shipments-of-akd1500-neuromorphic-processors/

[15] BrainChip Inc., "BrainChip developer resources: The MetaTF software development kit." BrainChip Developer, 2026. Available: https://developer.brainchip.com/

[16] BrainChip Inc., "Akida Engine Library." BrainChip SDK MetaTF documentation, 2025. Available: https://doc.brainchipinc.com/user_guide/engine.html

[17] BrainChip Inc., "Akida runtime API reference." MetaTF documentation, 2025. Available: https://doc.brainchipinc.com/api_reference/akida_apis.html

[18] K. D. Johnson, "Never idle: Scheduling a neuromorphic fleet with LSF and symphony." IBM Community, May 2026. Available: https://community.ibm.com/community/user/blogs/kevin-d-johnson/2026/05/28/never-idle-scheduling-a-neuromorphic-lsf-symphony

[19] N. Kotecha and R. Shrivastava, "Symphony-akida: On-chip fleet inference with IBM spectrum symphony and BrainChip akida." GitHub repository, 2026. Available: https://github.com/Brainchip-Inc/symphony-akida

[20] G. Lenz and D. McLelland, "Low-power ship detection in satellite images using neuromorphic hardware," *arXiv preprint arXiv:2406.11319*, 2024, Available: https://arxiv.org/abs/2406.11319

[21] K. D. Johnson, "Demonstrations." Personal website, 2026. Available: https://kevindjohnson.org/

[22] K. D. Johnson, "Velocity decoupled: The architecture of a symphony-enabled akida-based neuromorphic hive mind." IBM Community, May 2026. Available: https://community.ibm.com/community/user/blogs/kevin-d-johnson/2026/05/18/velocity-decoupled-symphony-akida-neuromorphic

[23] K. D. Johnson, "High performance quantum-centric supercomputing: A working implementation of heterogeneous orchestration across QPU, NPU, GPU, CPU, and other tiers." Technical paper, Mar. 2026. doi: 10.5281/zenodo.19888561.

[24] Intel Corporation, "Intel builds world's largest neuromorphic system to enable more sustainable AI." Intel Newsroom, Apr. 2024. Available: https://newsroom.intel.com/artificial-intelligence/intel-builds-worlds-largest-neuromorphic-system-to-enable-more-sustainable-ai

[25] D. S. Modha, F. Akopyan, A. Andreopoulos, *et al.*, "Neural inference at the frontier of energy, space, and time," *Science*, vol. 382, no. 6668, pp. 329–335, 2023.

[26] SchedMD, "Slurm workload manager – generic resource (GRES) scheduling." Slurm documentation, 2024. Available: https://slurm.schedmd.com/gres.html