AI Computing Broker: Optimizing AI Computing Efficiency

Authors: Matthias Loipersberger[1]*, Shinichi Awamoto[2], Godai Takashina[2] and Yusuke Nagasaka[2]§

Abstract

The rapid expansion of AI capabilities has driven enterprises and tech companies to invest heavily in GPU infrastructure. Despite these investments, GPU underutilization remains a significant challenge, with over 75% of organizations operating below optimal utilization levels. This inefficiency not only inflates infrastructure costs but also hampers AI development and innovation.

The AI computing broker (ACB) emerges as a pivotal solution to these challenges, offering dynamic GPU allocation, full memory access, and advanced scheduling algorithms. By monitoring AI workloads in real-time, ACB optimizes resource allocation, reduces idle time, and enhances throughput. This white paper provides technical leaders with an in-depth understanding of ACB's architecture, deployment strategies, and practical applications, demonstrating how ACB can transform AI infrastructure efficiency and drive competitive advantage.

[1] Fujitsu Technology Strategy Unit
[2] Fujitsu Research Computing Laboratory
Correspondence:* matthias.loipersberger@fujitsu.com §nagasaka.yusuke@fujitsu.com

Contents

1. Introduction

Enterprises and tech companies are making historic investments in GPU infrastructure to support rapid AI expansion. The hyperscalers are projected to **spend combined over $300B on AI infrastructure in 2025**, signaling a tectonic shift in corporate resource allocation toward compute-heavy systems. Furthermore, power consumption of large-scale AI infrastructure is projected to reach 10% of world electricity by 2030. Such investment underscores the transition from experimental AI to mission-critical production workloads, reflecting the strategic importance of high-performance computing.

Despite the massive scale of GPU investments, enterprises continue to struggle with underutilization; thus, directly eroding ROI. The State of AI Infrastructure at Scale 2024 report found that **over 75% of organizations operate below 70% GPU utilization even during peak times**. This inefficiency leads to bloated infrastructure spending, bottlenecks in model development, and constrained capacity to experiment with next-generation AI workloads. To remain competitive and extract full value from their AI investments, enterprises must urgently close the GPU utilization gap. Several factors contribute to suboptimal GPU utilization, including static job allocation, heterogeneous compute profiles, and inefficient scheduling. Static allocation ties up GPUs for the entire job duration, leading to significant idle time during CPU-intensive phases.

The AI computing broker (ACB) addresses these challenges through runtime-aware GPU allocation, full memory access, and advanced scheduling algorithms. It monitors framework-level activity to allocate GPUs dynamically, applies techniques like backfill to maximize throughput, and ensures active jobs receive the memory they need. These capabilities lead to higher GPU utilization, lower infrastructure costs, and faster AI development.

This white paper provides technical leaders with a clear overview of ACB—offering a look under the hood at its architecture, typical deployment patterns, and practical use cases. It explains how ACB integrates into common AI stacks and how it can be used to run more workloads with fewer GPUs. It is structured as follows:

- **Section 2** explores the root causes of GPU inefficiencies in contemporary AI workloads.
- **Section 3** introduces the AI computing broker, detailing its core components and technical design.
- **Section 4** outlines best practices to ensure successful implementation.
- **Section 5** describes the system architecture behind ACB.
- **Section 6** presents technical use cases: Running AlphaFold2 with heterogeneous compute profiles. Hosting multiple LLMs concurrently using ACB in conjunction with vLLM.
- **Section 7** demonstrates how ACB integrates into existing MLOps pipelines using Docker, Slurm, and Kubernetes.

2. The Root Causes of GPU Underutilization

GPU underutilization stems from static allocation, scheduling inefficiencies, and organizational friction. AI pipelines often alternate between GPU-intensive and CPU-bound phases, yet traditional schedulers reserve full GPUs for the entire job, leaving hardware idle during low-demand stages; this scenario is depicted in Figure *1* (a). Many pipelines are heterogeneous (see section use cases), making fixed allocation models ineffective. According to The State of AI Infrastructure at Scale 2024, **74% of enterprises are dissatisfied with current scheduling tools**, and only **19%** have visibility into job queues. Another related issue is **GPU memory partitioning** when attempting to share GPUs. Traditional approaches like GPU virtualization or multi-instance GPUs split the GPU's memory among multiple jobs, but this reduces the available memory per job. This limits the size of models or batches that each job can handle. This means that while virtualization can allow concurrent usage, each job is constrained by a smaller memory allotment, which can degrade performance or even prevent large models from running. These factors leave **over 75%** of companies operating below 70% GPU utilization at peak times.

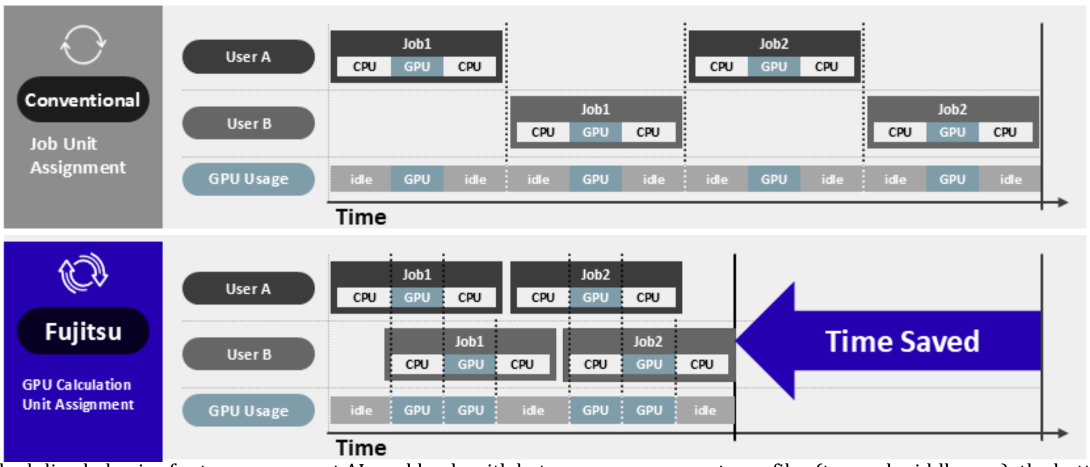

Figure *1*: GPU scheduling behavior for two concurrent AI workloads with heterogeneous compute profiles (top and middle row); the bottom row shows aggregate GPU utilization over time. (a) Without ACB: static GPU allocation leads to underutilization during CPU-bound phases. (b) With ACB: dynamic reassignment during idle GPU intervals improves overall efficiency.

3. AI Computing Broker Solution Introduction

The ACB delivers efficient GPU sharing for multiple AI applications through a dynamic, temporal sharing approach. Instead of statically partitioning GPU memory, ACB strategically swaps models between GPU and CPU memory based on real-time activity, enabling concurrent execution even for large language models (LLMs) exceeding the capacity of a single GPU. This runtime-aware allocation is managed by two key components: the GPU Assigner, a central scheduler, and the Adaptive GPU Allocator, a client-side library; the execution of concurrent AI workloads with ACB is illustrated in Figure *1* (b), for supported environments see Appendix A1 and for detailed competitive differentiation see Appendix B.

The GPU Assigner orchestrates GPU allocation using intelligent scheduling policies, including a backfilling mechanism. This allows smaller tasks to utilize available GPU resources even while larger tasks are queued, maximizing throughput and preventing smaller jobs from being blocked. The Adaptive GPU Allocator offers both automatic and manual modes. The automatic mode intercepts PyTorch API calls, transparently managing GPU assignments based on detected activity. The manual mode provides explicit control over allocation and release. This flexible approach streamlines integration and optimizes resource utilization based on actual application behavior.

Critically, ACB avoids the overhead and complexity of checkpointing by preserving application state during model swaps. This simplifies the execution of multiple concurrent experiments or model serving instances. By dynamically allocating GPUs, efficiently scheduling tasks, and providing flexible control over resource management, ACB empowers users to maximize GPU utilization, reduce infrastructure costs, and maximize the throughput of their AI workloads.

ACB Performance Overhead

The ACB system introduces minimal performance overhead in most AI workloads. A key design goal of ACB is to minimize impact on application performance while enabling efficient GPU sharing. Our measurements show that the cost of intercepting PyTorch calls, for example, is only ~5% in the context of AlphaFold2.

ACB's overhead primarily arises from two sources: data transfer between the host and devices when swapping models in and out of GPU memory, and communication with the central GPU Assigner via gRPC. The gRPC communication overhead is negligible. The overhead of hooks depends on model size, but negligible for considerably large models (see the estimation above for AlphaFold2). While large model transfers over PCIe can introduce noticeable latency, impacting individual job completion times, this cost is generally outweighed by the improvements in overall system throughput and GPU utilization that ACB provides. By efficiently allocating idle GPUs to pending jobs, ACB maximizes resource utilization, leading to more completed jobs in each period compared to a system without dynamic sharing. This translates to substantial performance gains at the cluster level, despite the potential increase in individual job latency due to data transfer. This trade-off is inherent in dynamic GPU sharing and represents a reasonable compromise given the significant benefits in overall resource efficiency.

4. Usage Guidelines

ACB delivers the highest value in environments with **heterogenous GPU workloads** and **multiple concurrent jobs**. The following best practices ensure optimal performance:

4.1 Best Practices for Maximizing ACB Impact

To get the most value from the ACB, it's important to understand the types of workloads and configurations where its dynamic GPU scheduling performs best. This section outlines key patterns including practical tips and guidelines. **The key questions are summarized in Infobox 1**, for more details see points A through F.

Infobox 1: Essential Questions for Evaluating ACB Suitability

1. Does your workload involve alternating phases of CPU and GPU tasks?
2. Do you have multiple GPU-capable jobs that can be queued or batched?
3. Is your workload capable of utilizing full GPU memory intermittently?
4. Do you need to host multiple domain-specific LLMs with uneven demand on shared GPU infrastructure?

A. Multi-Phase Workloads with Mixed CPU/GPU Profiles

ACB works best with jobs that alternate between GPU activity and meaningful CPU-bound work. This includes not only inference pipelines like AlphaFold2 but also training workloads with clear pre- and post-processing stages.

Important: Transferring task context between CPU and GPU memory is slow. To justify this overhead, CPU phases must be long enough to make the GPU handoff worthwhile.
Recommendation: The job should consist of more than 30% CPU-intensive tasks.
Tip: Use profiling tools (e.g., nsys, nvprof) to verify frequent CPU-only periods of at least several hundred milliseconds, see Appendix Figure 5 for a profiling screenshot.

B. Run Multiple Jobs to Fill the GPU Pool

ACB thrives when it can choose from multiple jobs waiting to use the GPU. If only one job is running, idle time can’t be reclaimed.
Tip: Maintain a queue or batch of GPU-capable jobs so ACB always has something to backfill.

C. Group Similar Jobs Together

ACB doesn't currently support preemption or prioritization, so it's best to co-schedule jobs with similar priorities. When jobs are too different, one may block resources or cause delays for others.
Example: Pairing two AlphaFold2 runs is ideal. Mixing a massive LLM job with a lightweight model may lead to poor balance.

D. Exploit Full GPU Memory

ACB enables full memory access per task rather than fractional assignment via MIG or vGPU. To capitalize on this, tasks should be capable of requesting full-GPU memory.
Rule of Thumb: Workloads that utilize 80–100% of GPU memory but intermittently release compute cycles benefit most from ACB's memory retention and task multiplexing.

E. Avoid Continuously GPU-Bound Jobs

Jobs with very high GPU utilization with no breaks leave no room for ACB to optimize, e.g. uninterrupted model training or dense inference workloads. The overhead from memory transfer and scheduling can even hurt performance in these cases.
Guideline: If the job is 90%+ GPU-active with minimal CPU-side work, ACB is not a good fit.

F. Single-Node vs. Multi-Node Deployment Strategies

Single-Node Deployment: Ideal for local testing or pilot workloads. Install the ACB package on a machine with multiple AI jobs and observe utilization improvements via monitoring tools.
Multi-Node Deployment: Suitable for production environments. ACB coordinates GPU allocation across nodes via shared scheduler hooks and runtime interceptors. Requires careful setting up of inter-node communication and persistent storage. See cluster deployment guidelines in the documentation.

4.2 Choosing the Right Scheduler:

ACB offers several scheduling policies optimized for different AI workloads;

Figure 2 illustrates how choosing the right scheduler can help improve GPU efficiency and model throughput.

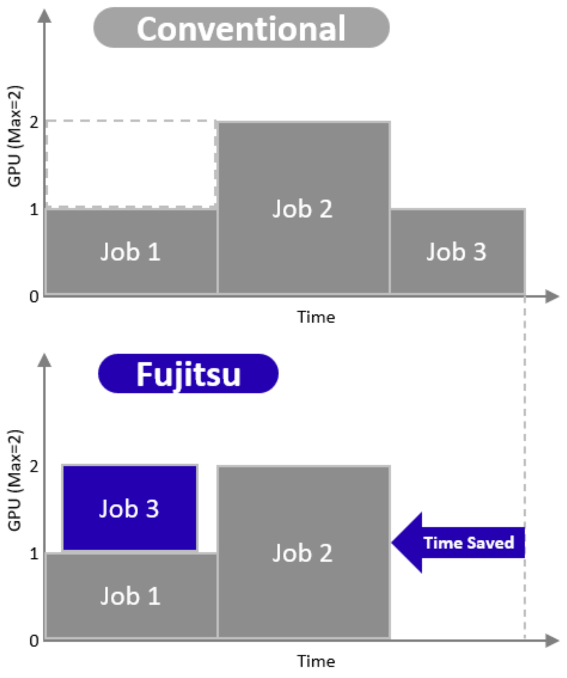

Figure 2:
Schematic of backfill scheduling impact on GPU job efficiency. The top timeline shows standard scheduling without backfill: Jobs 1 (1 GPU), 2 (2 GPUs), and 3 (1 GPU) are queued sequentially, causing idle GPU time. The bottom timeline shows backfill, where Job 3 is advanced to better utilize available GPUs.

The different algorithms are outlined below:

- **simple (default):** This first-in, first-out (FIFO) scheduler allocates one GPU to each job. Suitable for most single-GPU, single-node scenarios where jobs don't require GPU sharing.
- **gpu-sharing:** This scheduler enables spatial sharing of a single GPU among multiple jobs. Beneficial when individual AI models consume significantly less than half of the GPU memory. This scheduler maximizes GPU utilization by running multiple smaller models concurrently on the same GPU. Consider gpu-sharing when memory requirements per model are low but you have numerous models to run.
- **gpu-affinity (Multi-GPU Jobs):** This scheduler is designed for multi-GPU jobs and prioritizes efficient utilization. It operates on a FIFO basis but incorporates a "backfill" mechanism to enhance GPU usage. Backfill allows smaller jobs to "skip the line" and utilize free GPU resources even if larger jobs are waiting. For instance, if a 2-GPU job is queued but only one GPU is available, a smaller 1-GPU job can be scheduled to use the remaining GPU, maximizing resource use and potentially reducing overall job completion time. The scheduler also incorporates a GPU Affinity feature to avoid GPU migration and thus reduce context switching overhead.

5. AI Computing Broker Technical Architecture

This section outlines the core architecture of the ACB, which is designed with a modular, two-part structure:

- **Server-side:** A GPU assigner daemon
- **Client-side:** An ACB client, including a client library and a launcher script (*agarun*)

The GPU assigner operates as a daemon on GPU servers, tasked with discovering available GPUs and orchestrating their allocation to client applications. The ACB client integrates directly into your AI application process, enabling enhanced capabilities through the agarun launcher script. Communication between the ACB client and GPU assigner is facilitated via gRPC, ensuring efficient and reliable data exchange.

When an application requires GPU resources, it initiates an *AllocRequest()* gRPC call to the GPU assigner. Upon completion of GPU usage, the application sends a Release() gRPC call, allowing the assigner to reallocate resources to other applications. This per-request allocation model supports dynamic and flexible scheduling without interrupting ongoing processes. The architecture is illustrated in Figure *3*.

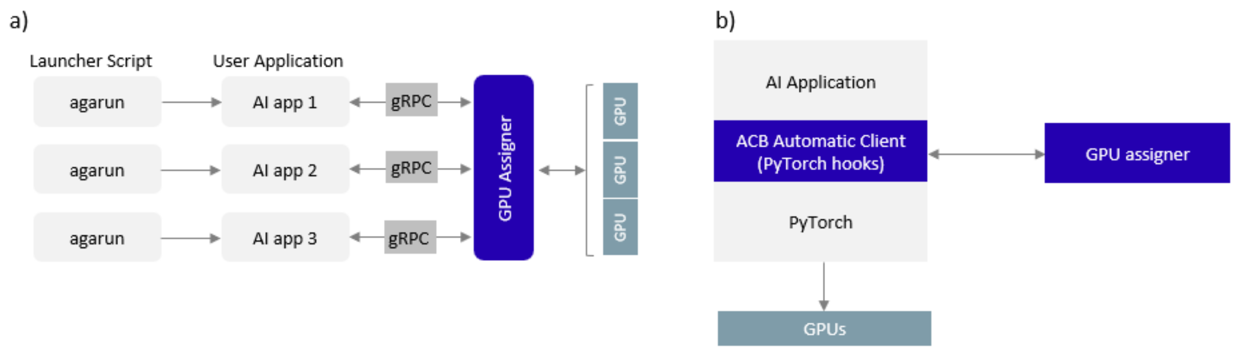

Figure 3: Architecture of a): the server-side: The GPU assigner daemon manages the GPU pool and allocates resources to applications via gRPC.; b) Client side: The ACB client intercepts PyTorch calls to monitor and analyze GPU usage patterns.

Multi-node Job Handling

ACB is equipped to handle multi-node jobs, currently limited to *torchrun*. In a multi-node configuration, GPU assigner services are deployed on each node, with one node designated as the controller and the others as executors. The controller node manages requests from all nodes and oversees the entire pool of GPU resources, while executor nodes focus on local GPU status monitoring, such as memory usage. Internal gRPC calls facilitate the collection of this information by the controller node, ensuring comprehensive resource management across the network (see Appendix Figure 6 for more information).

Automatic Client Integration

This functionality allows seamless integration of ACB into existing AI applications without code modifications. By running *agarun* with the automatic client enabled, *ACB PyTorch API* hooks are activated, detecting GPU device usage and communicating with the GPU assigner. The automatic client efficiently manages allocated GPU devices and computations on them, eliminating the need for manual tensor data movements. After GPU usage concludes, the client library automatically releases the GPUs, optimizing resource availability and minimizing idle time. Currently, the automatic client feature is exclusive to *PyTorch* applications.

Manual API Utilization

For scenarios requiring strict performance optimization, manual incorporation of ACB client API calls into application code is available. This method bypasses PyTorch API hooks, reducing overhead and enhancing efficiency. Developers can delineate GPU-utilizing code sections using *on_device_begin()* and *on_device_end()* APIs, allowing precise control over GPU requests and releases, as well as tensor data management.

Framework Support

The ACB client primarily supports the PyTorch framework, offering both manual API usage and automatic client integration. Basic support for TensorFlow is also provided, albeit without automatic client capabilities. Note that TensorFlow applications may experience process restarts when utilizing the ACB client. Summarized below:

Framework	Manual Mode	Automatic Client
PyTorch	Supported	Supported
TensorFlow	Limited	Not Supported

Infobox 2: For specialized application domains, developers can create custom client integrations by leveraging ACB client APIs, ensuring tailored or performance optimization.

6. AI Computing Broker in Action: Real-World Use Cases

To demonstrate the practical value of the ACB, we highlight two common AI infrastructure bottlenecks: multi-phase scientific inference and multi-LLM hosting under limited GPU budgets. These examples show how ACB delivers immediate efficiency gains—with no code changes and minimal integration.

6.1 Use Case 1: Maximizing AlphaFold2 Throughput with ACB

AlphaFold2 transformed structural biology by predicting protein 3D structures from amino acid sequences, solving a long-standing challenge and earning a Nobel Prize in 2024. However, its pipeline mixes CPU-heavy stages (e.g., MSA, template search) with GPU-intensive ones (e.g., Evoformer, Structure Module), resulting in idle GPU time.

Typically, two AlphaFold2 jobs run independently on single NVIDIA A100 GPU, reaching 12 proteins/hour. Due to static allocation, each job holds exclusive GPU access—even during CPU-bound stages—leading to significant underutilization. Live system metrics confirm low GPU usage despite reasonable throughput.

With ACB, several inference jobs share a single GPU dynamically. ACB monitors runtime activity, reclaiming idle GPU phases to serve the other jobs (8 in this example). This enables continuous activity via backfilling—without changing the AlphaFold2 script resulting in 32 proteins/hour.

Impact

- **+270% throughput improvement:** Achieved by switching from 2×A100 GPUs without ACB to 1×A100 GPU with ACB (8 concurrent jobs).
- **Performance gain:** Increased processing from 12 proteins/hour to 32 proteins/hour
- No code changes—drop-in deployment
- Lower cost and energy use by consolidating jobs

ACB lets bioinformatics teams run more AlphaFold2 jobs with fewer GPUs. This screening of larger molecular libraries helps teams make faster, better-informed decisions.

6.2 Use Case 2: Hosting Multiple LLMs on Shared Infrastructure with ACB

Enterprises increasingly deploy multiple domain-specific LLMs. Typically, each model uses a separate vLLM server pinned to a GPU, keeping weights in memory for fast responses. However, demand across models is uneven, and static assignment results in idle GPUs and high costs.

In standard setups, models like DeepSeek or Phi-4 are served by individual vLLM servers on dedicated GPUs. While low latency is ensured, this blocks resource sharing. Idle models occupy full GPUs, while high-demand ones can't scale.

ACB resolves this by managing multiple vLLM instances on shared GPUs. Instead of one server per model, ACB handles execution contexts and memory dynamically. By reclaiming idle memory and scheduling intelligently, it enables concurrent model serving—without delays or interference.

Beyond the 2-model demo, ACB scaled to over ten LLMs on 8 H100 GPUs. By optimizing memory fragmentation and dynamically adjusting GPU assignments, it enables dense multi-model serving.

Impact

- Fewer GPUs are needed for more models
- Eliminates model swap delays and cold starts
- Better memory usage and compute saturation
- Smooth inference across various LLMs

Strategic Implications

ACB changes the economics of multi-LLM hosting. For chatbots, multilingual support, or internal assistants, ACB enables elastic GPU sharing—cutting costs without sacrificing latency or performance. AI teams can scale horizontally (more models) and vertically (better GPU density).

6.3 Additional Verified Deployments

Beyond these core use cases, ACB has proven itself in trials with AI and cloud providers. Results show up to 2.25× efficiency improvements and substantial GPU idle time reduction; please refer to the press release for more details.

Representative Use Cases

- **FX Risk Models**: Improved training throughput for FX risk models via multiplexed GPU usage.
- **Cloud Services**: Expanded GPU access in cloud services to meet rising AI demand.
- **AI for Retail**: Boosted GPU utilization for AI camera systems used in retail.

7. Implementation Strategy

This section outlines how to deploy the ACB across a range of environments, from local testing to production-scale clusters. The goal is to provide technical teams with a clear path to adoption - whether running on a single machine or across distributed compute infrastructure. For more detailed information see Appendix A2 and A3.

Deployment Modes

ACB supports a flexible set of environments to meet the needs of both research and enterprise teams:

- **Bare Metal** – Direct installation on Linux servers with supported GPU drivers.
- **Docker Containers** Pre-configured container images for rapid deployment.
- **Job Scheduler Integration** – Works with SLURM and other common schedulers to orchestrate AI workloads in shared environments.
- **Kubernetes (Planned)** – Future versions will support dynamic scheduling across containerized clusters. Refer to release notes for status.

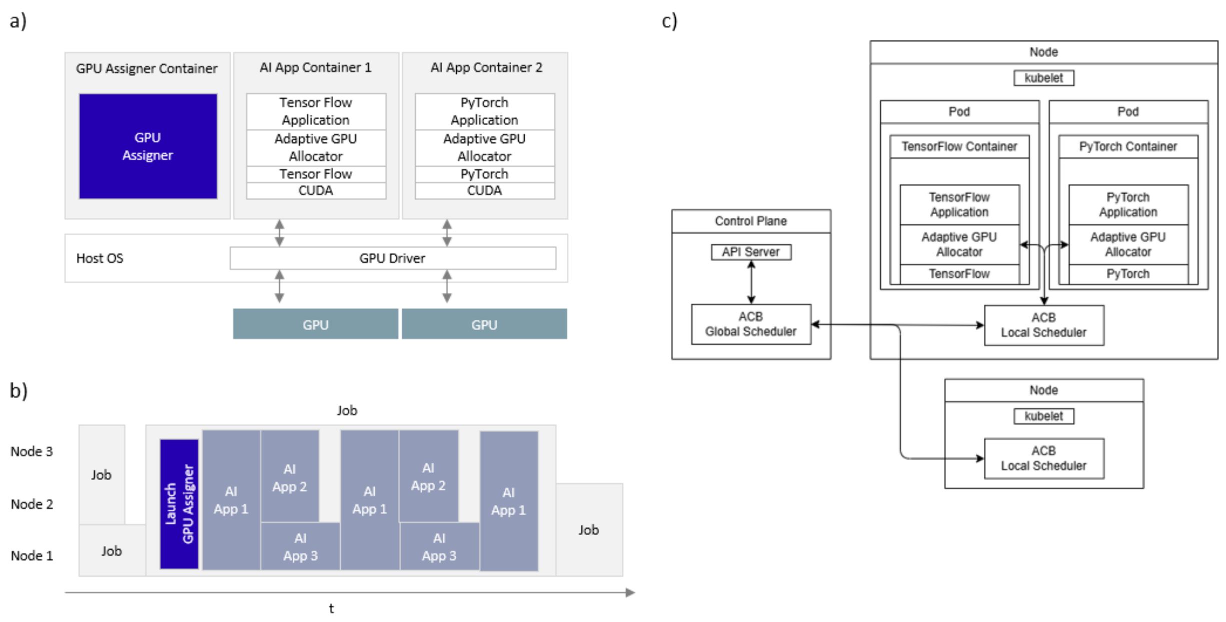

Figure 4: Integration of ACB with (a) Docker: ACB uses Docker to isolate and coordinate its GPU Assigner and Allocator for efficient resource use. (b) Slurm: ACB enhances GPU sharing in Slurm via batch scripts without modifying Slurm itself. (c) Kubernetes: ACB supports fine-grained GPU sharing in Kubernetes with a dual-scheduler design.

7.1 Deploying ACB with Docker

ACB utilizes Docker to streamline deployment and management of its core components: GPU Assigner and Adaptive GPU Allocator. Each component runs in separate containers, communicating over a dedicated Docker network. The GPU Assigner operates as a standalone container, functioning like a system service, and awaits resource requests from client applications. User AI programs are deployed in individual containers, where the Adaptive GPU Allocator automatically loads the program and communicates with the GPU Assigner via the Docker network.

7.2 Slurm Integration

ACB can be deployed within a Slurm-managed cluster for enhanced GPU resource management without modifying Slurm itself. Users benefit from Slurm's node allocation and ACB's efficient intranode GPU sharing. Deployment involves adding startup commands for the GPU Assigner and Adaptive GPU Allocator in Slurm batch scripts. Upon job launch, Slurm allocates nodes, and the batch script initiates ACB components. The GPU Assigner detects available GPUs on each node, and a configuration file specifies IP addresses for inter-node GPU management. This setup allows multiple AI applications to share GPUs within Slurm workloads.

7.3 Kubernetes Integration (Coming Soon)

ACB addresses GPU resource management challenges in Kubernetes by integrating fine-grained GPU scheduling. The Local Scheduler, deployed as a Daemon Set, ensures presence on nodes with GPUs, coordinating with ACB’s Global Scheduler for efficient resource distribution. Within each Pod, the ACB client library interacts with the Local Scheduler to manage GPU resources, allowing temporal sharing among applications. This integration leverages the NVIDIA GPU Operator for seamless hardware interaction, enhancing GPU utilization and simplifying management. Note that ACB's "backfill" feature is unavailable in Kubernetes integration, with jobs scheduled FIFO.

Summary

The ACB represents a transformative approach to addressing the pervasive issue of GPU underutilization in enterprise AI infrastructure. Despite historic investments in GPU resources, many organizations struggle to achieve optimal utilization due to static allocation models and inefficient scheduling practices. These challenges lead to inflated costs and hinder the ability to innovate and scale AI workloads effectively.

ACB tackles these inefficiencies by introducing a dynamic orchestration layer that intelligently allocates GPU resources based on real-time workload demands. This approach not only maximizes GPU utilization but also reduces infrastructure costs and accelerates AI development. The white paper illustrates ACB's impact through detailed use cases, such as enhancing AlphaFold2 throughput and enabling multi-LLM hosting on shared infrastructure. These examples demonstrate how ACB can deliver substantial performance improvements and cost savings without necessitating code changes.

ACB's flexible deployment options, including bare metal, Docker, Slurm, and future Kubernetes integration, cater to diverse operational environments, ensuring seamless adoption and scalability. By implementing ACB, organizations can unlock the full potential of their GPU infrastructure, driving faster AI development and improved ROI.

Call to Action

As AI workloads scale and GPU supply remain constrained, **underutilized infrastructure has become a hidden tax on innovation**. The ACB turns this inefficiency into a strategic advantage, unlocking higher throughput, lower cost per model, and faster iteration across AI pipelines.

Now is the time for AI leaders to act: Integrate ACB into your existing stack, benchmark its impact, and reclaim performance from hardware you already own. Organizations that optimize GPU utilization today will be the ones scaling AI tomorrow—faster, leaner, and ahead of the curve.

Acknowledgement

We would like to thank I-Hsi Kao for providing valuable technical review and insights. We are also grateful to Ryuji Iwata for his thoughtful support in refining the overall formatting, design, and diagrams.

Appendix A: System Compatibility & Integration Scenarios

To ensure successful deployment of the ACB, this appendix outlines its current system compatibility and highlights advanced integration scenarios. The ACB middleware is designed to be lightweight and flexible, with minimal infrastructure overhead and no required changes to model code.

A1 – Supported Environments

Component	Compatibility
GPU Drivers	NVIDIA CUDA-compatible (v11.x–12.x); tested with CUDA 11.8+ and driver 525+
Operating Systems	Ubuntu 20.04 / 22.04
AI Frameworks	PyTorch (2.1.2+): native hooks used to detect GPU usage, TensorFlow (2.15+): limited support in manual-var mode.
Hardware Support	NVIDIA A100, H100, L40S, A10; supports MIG-enabled and non-MIG GPUs (MIG support coming soon)
Inference Runtimes	Compatible with vLLM,
Containerization	Docker supported; no dependency on Kubernetes
Cluster Integration	Single node supported; multi-node support limited until Q3/2025
Licensing / Deployment	Python package; license validation via API

A2: Integration Notes

- **Model Transparency**: ACB does not require any changes to user models or training scripts if PyTorch is used as the AI framework. It observes GPU activity through backend runtime instrumentation via framework profiling APIs.
- **Memory Handling**: ACB temporarily swaps entire models in and out of GPU memory based on runtime activity, allowing for oversubscription scenarios where all models combined exceed total memory available.
- **Execution Context**: The broker can wrap a model training or inference job and dynamically release the GPU when idle phases (e.g., CPU preprocessing) are detected. Full memory access is granted during active GPU use.

A3: Advanced Compatibility Scenarios

Scenario 1: ACB + NVIDIA MIG (Driver-Level Partitioning)

When deployed on MIG-enabled GPUs (e.g., A100 or H100), ACB complements MIG by dynamically assigning MIG instances to jobs in real time. While MIG creates isolated GPU slices (e.g., 10 GB or 20 GB partitions), ACB ensures these slices are only bound to jobs during their active GPU phases. This allows multiple GPU-bound jobs to co-exist with minimal memory conflict or underutilization (MIG support coming soon).

Scenario 2: ACB + Triton Inference Server / vLLM (Application Layer)

Inference-serving frameworks like Triton and vLLM typically launch persistent GPU-bound workers. When integrated with ACB, inference models are loaded on-demand into GPU memory and unloaded post-inference, enabling **high-throughput LLM serving** even with limited memory -ideal for bursty or multi-model deployment scenarios (currently only vLLM support).

Scenario 3: ACB + NVIDIA MIG + vLLM (Full Stack Efficiency)

Combining all three technologies creates a powerful tiered system:

- **MIG** partitions a single GPU into multiple isolated slices—ideal when models don't need the full GPU.,
- **ACB** operates within each MIG slice, dynamically orchestrating multiple LLM models.
- **vLLM** enables fast, efficient LLM inference.

This architecture supports **dense multi-model hosting**: small and mid-sized models can share a MIG slice, improving utilization without contention.

Appendix B – Competitive Differentiation: ACB's True Innovation in GPU Efficiency

We assessed Fujitsu's **ACB** against three leading platforms—**Run:ai**, **Exostellar.ai**, and **Slurm**—to validate the comparative feature matrix and highlight where ACB stands out.

Feature Verification Table

Feature	**ACB**	**Run:ai**	**Exostellar.ai**	**Slurm**
Runtime-Aware Scheduling	Yes	No: schedules at container/job granularity time slicing	No: monitors and dynamically adjust resource allocations	No: static allocation per job, no intra-job rescheduling
Memory Partitioning	Yes – gives full GPU memory at each active phase	Yes – fractions and automated MIG partitions	Yes – Software-Defined GPU virtualization	Yes – via MIG and software "sharding"
Memory Oversubscription	Yes – via time-based trading of full GPU memory	No – avoids memory oversubscription per GPU for safety	Yes – supports safe over-commit with virtual devices	No – not supported; shards are static, MIG is partitioned
Cluster-Level Orchestration	Planned;	Yes – full Kubernetes integration	Partial – integrations via K8s; includes migration and scaling	Yes – platform in HPC clusters; well-defined queue and scheduling
Plug-and-Play Integration	Lightweight – middleware, no code changes	Moderate – requires K8s + Run:ai deployment	Moderate – system-level driver/intel install on each node	Heavy – full HPC scheduler setup
Key Differentiator	Task-level GPU capture	Dynamic GPU fraction and resource quota management.	Software GPU virtualization + oversubscription commit	Batch scheduling with high reliability & scale

Competitive Highlights

ACB

- Dynamically assigns **full GPU memory and compute** only when a task enters GPU phases (e.g., forward/backpropagation), identified by runtime behavior. It minimizes hardware contention throughout a job's lifetime.
- Enables memory **oversubscription** by treating GPU memory as fully reusable across tasks—users demonstrated handling 150 GB of AI processing on 40 GB GPUs via time-based swapping.

Run:ai

- GPU fraction and Dynamic MIG feature dynamically divide GPU memory at the time when a job is submitted.
- Offers sophisticated resource quota allocation based on Projects and Departments, allowing fair scheduling and temporal over-quota.
- Focusing on Kubernetes cluster environment and offering tight integration with it.
- It does not handle memory oversubscription -- it does not allow concurrently running two jobs each demanding entire GPU memory (see here).

Exostellar.ai

- Employs **Software-Defined GPU (SDG)** virtualization: tasks get isolated virtual GPUs with configurable memory/compute.
- Allows safe **memory oversubscription** using paging and virtual memory swapping when tasks don't simultaneously use peak resources.
- Monitors and dynamically adjusts resource use via telemetry; includes Kubernetes integration and clustering tools for migration and autoscaling (see here).

Slurm

- The de facto **HPC scheduler**, offering stable, policy-driven job queueing and MIG/shard support for partitioned GPU use.
- Introduced **software GPU sharding** (v22.05), allowing fractional job allocation—**but without memory fencing**; users must ensure safe co-scheduling.
- Lacks **runtime GPU reallocation**; jobs keep GPU allocation throughout their lifespan.

Appendix C: Additional Information

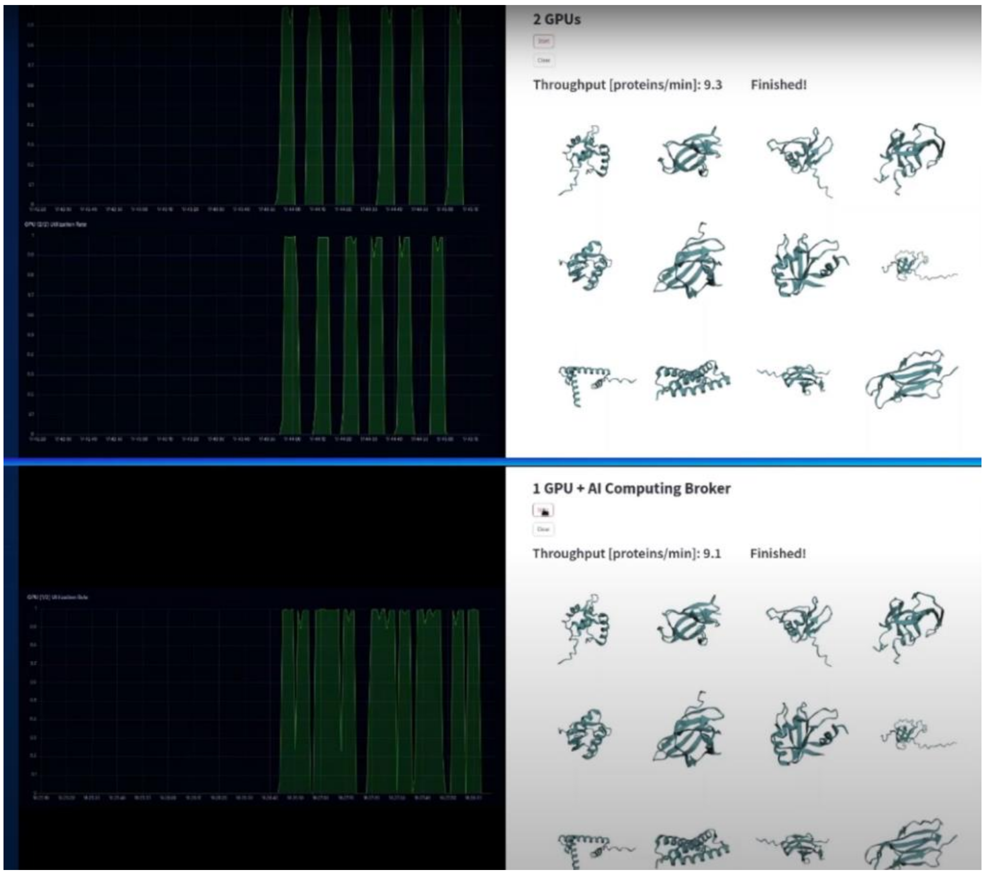
2 GPUs
Throughput [proteins/min]: 9.3
Finished!
1 GPU + AI Computing Broker
Throughput [proteins/min]: 9.1
Finished!

Figure 5: Top: Two GPUs running Alphafold2 without ACB; bottom: a single GPU running the same two Alphafold2 jobs on a single GPU. Left side: GPU activity, right side: output of the Alphafold2 inference.

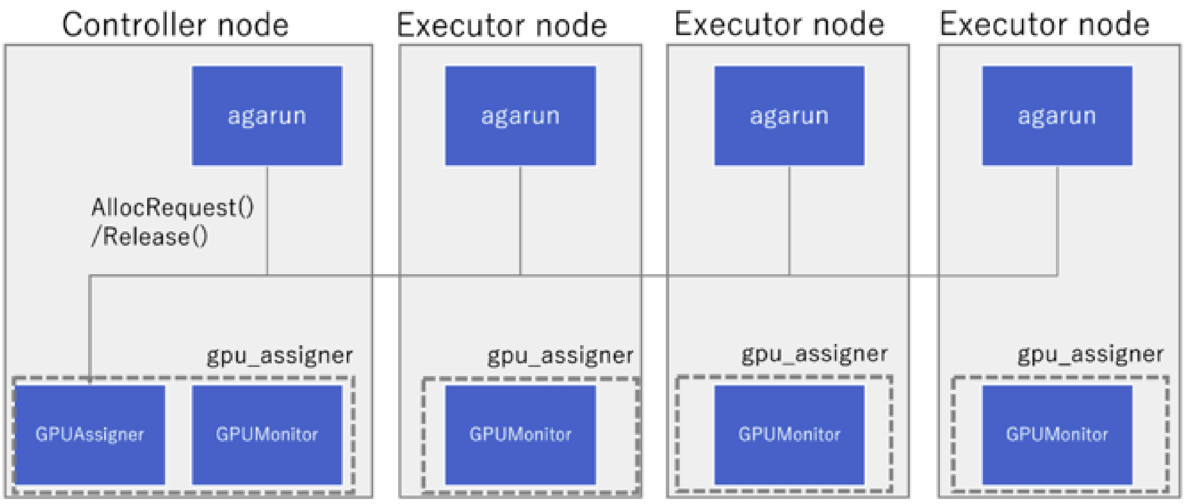

Figure 6: Architecture of the ACB multi-server set up: The main GPU assigner is located at the control node and interacts with GPU monitoring agents on each executor node to control the global GPU resource pool.