ChatGPT

Dynamic CPU-GPU Scheduling for AI Workloads – Market & Competitive Landscape

Introduction

Building a **dynamic runtime CPU-GPU routing and scheduling layer** aims to boost compute utilization and reduce idle GPU time in AI workloads. The idea involves analyzing a model's computation graph (offline or ahead-of-time) and then making **runtime decisions** on whether each operator should execute on CPU or GPU, based on input size or workload characteristics. Longer-term, this could enable features like dynamic kernel switching (choosing the best implementation on the fly), overlapping data movement with computation, and co-scheduling multiple models on the same hardware. This report provides a comprehensive analysis of existing solutions in this space, evaluates market saturation and competitive gaps, and discusses strategic focus areas, potential pitfalls, and opportunities for a new entrant. Finally, we assess the overall chance of success and recommend a high-impact initial problem to tackle.

Existing Solutions and Approaches

1. Deep Learning Frameworks (PyTorch, TensorFlow): Mainstream frameworks typically do *not* offer automatic per-operator CPU/GPU scheduling. In PyTorch or TensorFlow, if a model or tensor is moved to GPU, all supported operations run on GPU by default, leaving CPU largely idle [1] [2]. PyTorch developers have discussed a "hybrid" mode to use CPU when free, but warned that the overhead of moving data between CPU and GPU often outweighs any benefit [2]. In practice, frameworks rely on static device placement or user-defined placement (e.g. manually moving certain submodules to CPU). Some **TensorFlow** deployments do place specific small ops on CPU (especially ops with no GPU kernel or where GPU has no benefit), but this is done via static heuristics. Overall, base frameworks have **no general dynamic scheduler** that shifts ops between CPU and GPU at runtime for utilization reasons – integration complexity and data transfer overhead have been deterrents.

2. ONNX Runtime (Microsoft) and OpenVINO (Intel): ONNX Runtime supports **execution providers (EPs)** for different devices (CUDA, TensorRT, CPU, etc.) and can perform *heterogeneous execution*. In a typical scenario, ONNX Runtime will run most ops on GPU but *fallback* to CPU for any ops not supported on the GPU EP [3]. This fallback ensures model execution succeeds, but it can hurt performance when it triggers, due to the cost of data transfer between devices [3]. Notably, ONNX Runtime may even choose to execute certain ops on CPU **for performance reasons** – it has an internal cost model to decide placement in some cases [3]. Intel's **OpenVINO** goes further by offering a *Heterogeneous (HETERO) plugin*, where a model can be partitioned across GPU and CPU. OpenVINO can automatically assign layers to GPU or CPU and allows manual affinity overrides [4] [5]. This is largely a compile-time or load-time decision: e.g. "HETERO:GPU,CPU" means try GPU first, then CPU [4]. It addresses GPU memory limits by splitting models, or offloading unsupported ops. However, OpenVINO documentation cautions that splitting too many subgraphs between devices can be counterproductive if data transfer overhead outweighs benefits [6]. In sum, ONNX Runtime/OpenVINO do support *per-op routing* but mostly as a static or fallback mechanism, not a continuously adapting runtime scheduler based on workload dynamics.

3. NVIDIA Tools and Inference Servers: NVIDIA's software stack doesn't explicitly provide a per-operator CPU/GPU scheduler, but it offers ways to maximize GPU utilization. The **Triton Inference Server** allows deploying multiple models (or multiple instances of the same model) concurrently on a GPU [7]. It supports dynamic batching and concurrent model execution to keep the GPU busy [8]. For example, if two requests for different models arrive, Triton can schedule both on the GPU in parallel (using separate streams or context switching) [9]. This improves throughput and utilization without splitting individual operations across devices. Triton can also serve models on CPU or GPU, but *not* split one model's ops between CPU and GPU automatically. NVIDIA's **CUDA** environment supports overlapping computation and data transfers via streams, which skilled developers use to hide data movement latency. NVIDIA also introduced **MIG (Multi-Instance GPU)** to partition a GPU for multi-tenant usage, and tools like MPS (Multi-Process Service) for time-slicing GPU among processes – again, these increase overall utilization, but at the job level rather than the operator level. There are also specialized libraries: e.g. **cuDNN** (for neural net ops) internally chooses different algorithmic kernels based on input size (which is a form of dynamic optimization, albeit purely on GPU). **TensorRT** (for optimized inference) focuses on GPU kernels and requires that the model fits on GPU; if an op is not supported, it typically requires a fallback path (possibly via ONNX Runtime CPU, which is not seamless). In summary, NVIDIA's focus has been on *concurrent execution and efficient GPU kernels*, not so much on leveraging CPU in parallel. An exception is data preprocessing: NVIDIA DALI can use GPU for data augmentations to offload CPU (the inverse of our focus).

4. Model Compilers (TVM, Glow, etc.): Apache TVM compiles models to optimized code for specific hardware targets (CPU, GPU, etc.). By design, TVM could compile different parts of a model for different devices, but this would be a **static partition** determined at compile time. There isn't a mainstream TVM feature that at runtime evaluates input shapes to switch device – instead, one could compile multiple versions or include conditionals. **Facebook's Glow** compiler explicitly supports *heterogeneous partitioning* of a neural network graph [10]. Given a list of devices (e.g. one CPU, one GPU), Glow can partition the model into segments assigned to each device [10]. The partitioning can be guided by heuristics or cost models (for example, to fit memory constraints or balance compute load). This is again determined before execution; Glow's aim is to produce an execution plan that might run, say, certain layers on CPU and the rest on GPU. It does not dynamically change the plan per inference. These compiler-based approaches show that splitting work across CPU/GPU *is feasible*, but they typically create a fixed schedule. They excel at ahead-of-time optimization (and can incorporate profile-guided decisions), but lack a runtime scheduler to adapt on the fly to each input or system state.

5. Distributed Computing Frameworks (Ray, Dask, etc.): At a higher abstraction, tools like **Ray** allow users to schedule tasks or model inference on different resources (some tasks on CPU, some on GPU) easily. **Ray Serve**, for instance, can route requests to a CPU worker or a GPU worker. However, this is *coarse-grained*: it might route an entire model inference to CPU vs GPU based on availability or load, but it does not split individual model layers between devices. Ray's scheduling is at the level of tasks/actors across a cluster, not internal to a single model execution. Similarly, frameworks like **Apache Arrow** provide a zero-copy memory format and can facilitate moving data between CPU and GPU processes (Arrow has CUDA support for sharing device memory). This can help reduce overhead when CPU and GPU tasks are pipelined, but Arrow itself is not a scheduling system. It's more of an enabling technology for efficient data movement, which could be used *within* a scheduling solution to minimize transfer costs.

6. AI Infrastructure Startups and Platforms: There are a few players focusing on **GPU utilization and scheduling** in a broader sense: - **Run:AI (NVIDIA)** – A platform for GPU virtualization and scheduling in Kubernetes. Run:AI introduced **Dynamic GPU Fractions**, allowing jobs to use a fraction of a GPU and burst above that if the GPU has free capacity [11] [12]. This addresses the common situation where many workloads (especially inference workloads) do not fully utilize the GPU they reserve [11]. By treating GPU memory and compute as shareable resources, multiple jobs can co-exist, each getting a guaranteed portion and using extra during idle periods [12]. This is effectively *time-sharing or space-sharing at the job level* (multiple workloads on one GPU). While not a per-operator router, it tackles the **same core problem (low utilization)** via multi-tenant scheduling. - **Determined AI / MosaicML / OctoML:** These focus on optimizing training or inference performance. For instance, OctoML uses compile-time optimization (via TVM) to make models run faster on given hardware, but not specifically dynamic CPU offloading. MosaicML (now part of Databricks) and others provide inference optimizations (quantization, etc.) and efficient scheduling in clusters, but again nothing known for fine-grained CPU/GPU routing within a single model. - **Cloudflare's Infine Engine (Infire):** Cloudflare recently built a custom inference engine "Infire" in Rust to maximize utilization of their edge GPUs [13] [14]. One issue they encountered with existing solutions (like vLLM) was the inability to **co-host multiple models on the same GPU** concurrently [14]. Infire introduces a dynamic scheduler that can run multiple different models on one GPU, minimizing downtime between requests [14]. This addresses multi-model **co-scheduling**, allowing better utilization especially on smaller edge GPUs that might be underutilized by a single model. Cloudflare's engine also reduces CPU overhead (by using Rust instead of Python) and integrates research ideas (like continuous batching). While Infire is in-house and LLM-focused, it's a notable example of an **industry solution to dynamic scheduling** – focusing on overlapping multiple models and requests rather than offloading to CPU, but with a similar goal of higher utilization. - **Others:** No startup has a clear lead in "per-operator CPU/GPU scheduling" as a standalone product. Some research spin-offs might exist (for example, **EdgeCortix** or **Deeplite** work on model optimization, though not specifically hybrid scheduling). **Neural Magic** focuses on CPU-only inference efficiency (leveraging CPU sparsity and threading) – essentially the opposite approach (avoiding GPUs). **Petals** and others focus on distributed inference across machines. In general, the space of **GPU utilization optimization** is active, but solutions tend to cluster either at *the cluster level (schedulers like Run:AI, Kubernetes-based autoscalers)* or *the model optimization level (compilers, quantization, batching)* rather than at the **fine-grained CPU-GPU collaboration level**. This suggests that the concept of a dynamic CPU/GPU scheduling layer per model is mostly being explored in academia and internal projects, not yet a crowded commercial product category.

7. Research Prototypes and Academic Work: In the last few years, numerous research efforts have tackled hybrid CPU-GPU execution for AI: - **CoDL (2022)** – An efficient CPU-GPU co-execution framework for deep learning *inference* on mobile devices [15]. CoDL dynamically schedules operators on CPU or GPU by evaluating each operator's affinity for the processors, aiming to minimize latency [15]. It proved that concurrent execution on heterogeneous processors can speed up inference by utilizing both CPU and GPU on a phone. CoDL achieved modest gains (e.g. ~11% speedup in some cases) and highlighted challenges like balancing loads. - **Band, BlastNet (2022)** – These systems tackled scheduling for *multiple DNNs concurrently* on edge devices. **Band** (Jeong et al. 2022) and **BlastNet** (Ling et al. 2022) can partition and schedule multiple models across processors, e.g., running some models on CPU, some on GPU concurrently to meet latency Service Level Objectives. They are more about multi-DNN task scheduling than intra-DNN operator scheduling [16]. - **NN-Stretch (Wei et al. 2023)** – A framework that can *automatically branch or trim models* for different hardware capacities (not exactly CPU/GPU scheduling, but related to adaptive inference). - **SparOA (Zhang et al. 2025)** – A recent edge-focused hybrid inference framework that explicitly considers each operator's **sparsity** and **computational intensity** when scheduling [17] [18]. SparOA uses a lightweight offline predictor to determine thresholds, and a reinforcement-learning-based runtime scheduler to allocate ops to CPU or GPU dynamically [17]. The insight is that some ops (like convolutions or GEMMs) have high computational intensity and run best on GPU, whereas others (like activation or normalization layers) are low-cost and can run on CPU with little penalty [19]. If an operator is very sparse or small, CPU might handle it efficiently, freeing the GPU or allowing parallel overlap. SparOA showed up to **50.7% speedup over CPU-only execution** and outperformed prior static co-execution methods by adapting to runtime conditions [20]. It also saved 7–16% energy versus the best static baseline [21]. SparOA's evaluations on NVIDIA Jetson (edge GPU) demonstrated significant latency reductions by dynamic scheduling [22] . - **APEX (Fan et al. 2023)** – A hybrid CPU-GPU scheduler for **LLM inference**. Large Language Model decoding is often *memory-bound* due to the growing KV cache, and prior approaches offloaded parts of the model to CPU (like the attention mechanism) but suffered from poor overlap [23] [24] . APEX uses profile-informed scheduling to predict CPU vs GPU sub-task times and overlaps CPU work (like attention with CPU-managed KV cache) with GPU work, without needing to split batches [25] [26] . It **improved throughput by 1.1× to 1.9×** over GPU-only baseline on certain GPUs, while preserving latency [26] . Against other hybrid-offload solutions, it delivered up to ~49% higher throughput [26] . This is a cutting-edge example of fine-grained scheduling tailored to LLM decode phases. - **RecOS (IJCAI 2025)** – A system for **concurrent recommendation model inference** on GPU, focusing on inter-operator scheduling across queries. It identifies that naive first-come-first-serve execution of ops from multiple concurrent queries underutilizes the GPU and causes contention [27] [28] . RecOS monitors the GPU workload and assigns ops from different queries to different CUDA streams to exploit inter-op parallelism [29] . It introduced an asynchronous tensor management to ensure correctness across streams. In evaluations, RecOS reduced tail latency by up to **68%** under high concurrency [30] . This is more about multi-stream scheduling on GPU (not offloading to CPU), but it shares the goal of better GPU utilization and could be complementary to CPU-offloading ideas. - **FusionFlow (VLDB 2023)** – A system targeting **training data preprocessing**. It "harnesses idle GPU cycles" to offload part of the data prep work to the GPU while the GPU would otherwise be waiting for data [31] [32] . FusionFlow dynamically splits each batch's augmentation tasks between CPU and GPU, feeding larger image transformations to the GPU and smaller ones to CPU in parallel [33] . By overlapping data prep with training, it speeds up training by 46–91% compared to GPU-only preprocessing (NVIDIA DALI) [34] . This highlights how *idle time overlap* can yield big wins in training scenarios where the GPU often waits for the next batch. The authors consciously chose to partition at the data level rather than operator level for preprocessing, citing that treating entire small batches separately for CPU/GPU was simpler and more effective in that context [35] .

8. Other Noteworthy Efforts: For completeness, it's worth noting **DeepSpeed (Microsoft)** tackled a related problem in training: **ZeRO-Offload** moves parts of training computation (optimizer steps, gradient updates) to CPU to save GPU memory, and crucially overlaps those CPU computations with the next forward/backward pass on GPU [36] . This hides the CPU cost by delaying parameter updates by one iteration [36] . The result is minimal performance loss despite using CPU for some work, proving that clever scheduling can keep the GPU busy and avoid idle gaps even when using CPU for heavy tasks [37] . While ZeRO-Offload addresses memory scaling more than utilization, it exemplifies *overlap to reduce idle time*, a principle very relevant to our idea.

Summary of Landscape: In broad strokes, **large companies and frameworks** provide partial solutions (static splitting in OpenVINO, multi-model concurrency in Triton, memory offload in DeepSpeed) but **no dominant player offers a general dynamic CPU-GPU scheduler at operator granularity**. Several **research projects** have demonstrated the concept's value in specific domains (edge vision models, LLMs, recommender systems). **Startups and cloud platforms** are targeting GPU utilization through other means (multi-tenant sharing, better compilers, or custom engines), indicating that the **market recognizes the cost of idle GPU time**. However, the lack of an out-of-the-box, flexible runtime scheduling layer suggests *commercial white space* for a solution that generalizes these ideas.

Market Saturation and Gaps

Inference vs. Training: The **inference optimization space** is active and somewhat saturated in certain areas. For example, LLM inference has many solutions focusing on throughput and memory (transformer-specific optimizations like vLLM, FasterTransformer, DeepSpeed-Inference, Hugging Face Accelerate offload, etc.). High-throughput inference servers (Triton, TorchServe, NVIDIA NeMo) let companies utilize GPUs efficiently by batching and concurrent execution. That said, these typically assume the heavy lifting stays on GPUs – few address using CPUs in parallel except for memory offload scenarios. **Training** optimization is also a crowded field (distributed training frameworks, mixed precision, pipeline parallelism, etc.), but specifically *reducing GPU idle time during training* is less of a marketed feature. Many training pipelines still suffer from GPU stalls waiting for data, implying an opportunity that only specialized research (like FusionFlow) has touched. So, **inference at scale (especially for large models)** has partial solutions (market somewhat warm), whereas **training utilization and fine-grained scheduling** are less covered in products (more white space).

Edge vs. Cloud: On **edge devices**, the need for CPU-GPU hybrid execution is **pressing** – devices like NVIDIA Jetson have relatively weaker GPUs and moderately strong CPUs, and workload sizes vary. The research community (SparOA, CoDL) has focused on edge use-cases, but commercially, edge runtime libraries (TensorFlow Lite, CoreML, etc.) do more model pruning/quantization than dynamic scheduling. There is not a well-known edge inference engine that does operator-by-operator scheduling between CPU/GPU; most edge deployments run either fully on the GPU (if available) or CPU, or use the GPU for what it can and CPU only as fallback. In the **cloud**, the trend is to throw large GPUs at the problem or run many jobs concurrently. Cloud customers with GPU clusters often care about utilization, but they use orchestration (job schedulers, autoscalers) to keep GPUs busy. A dynamic per-op scheduler could still help in cloud scenarios for *specific models* (e.g. LLM decode offload as in APEX, or CPU handling data prep), but it's less expected by default – typically, one big model saturates a cloud GPU fully, or multiple jobs share it. That said, there is underexplored territory in cloud **multi-model co-location** (Cloudflare's work is a rare example), and in **automating overlaps** that cloud practitioners currently implement manually (like overlapping data transfers or using CPU threads for pre/post-processing).

General AI Models vs. LLMs: LLMs (Large Language Models) have spurred innovations in scheduling because their *memory footprint and iterative nature* cause unique bottlenecks. Techniques like splitting attention to CPU, or continuous batching, are niche solutions aimed at LLM serving. This area is heating up, with several teams working on maximizing LLM inference throughput (so one could argue aspects of LLM-specific scheduling are becoming saturated – many open-source projects tackling caching, offloading, concurrency for LLMs). In contrast, more **general models (CV, audio, multimodal)** haven't seen as much focus on CPU-GPU hybrid execution, perhaps because traditional CNNs or transformers for vision either fit well on GPUs or run on CPUs when small. There may be a gap in optimizing those *mid-sized models*: for example, a ResNet that doesn't fully utilize a modern GPU could potentially run layers concurrently on CPU to handle more throughput or reduce latency, but few if any frameworks attempt this. Likewise, **multimodal pipelines** (where part of the pipeline is vision, part language, etc.) could benefit from smarter device utilization (e.g. run the vision CNN on GPU, but generate text on CPU if GPU is busy). This remains underexplored; most pipeline orchestrators run each model sequentially on GPU if possible.

Dominant Players and Saturation: Overall, **no single company "dominates" the exact space of dynamic CPU/GPU scheduling**. NVIDIA and Google dominate GPU-serving infrastructure in general, but their solutions (Triton, TF Serving, etc.) do not incorporate dynamic CPU offloading beyond static assignments. Startups like Run:AI dominate cluster-level GPU efficiency management; their presence means the general **problem of GPU underutilization** is recognized and solutions exist (so customers are aware of the issue). However, those solutions address it by *sharing GPUs between jobs* rather than improving a single job's device utilization. This implies a **white space** for solutions that optimize at a finer granularity – potentially complementary to what Run:AI does. In summary: - **Cluster/ Multitenancy level** – moderately saturated (several solutions). - **Compiler level** – moderately active (TVM, etc., but they produce static solutions). - **Runtime operator level** – relatively open, mostly in research or specific frameworks (OpenVINO) and not widely deployed.

Areas like *multi-model co-scheduling on one GPU* and *real-time CPU offload* remain niche. Even Cloudflare had to build their own engine, indicating the lack of an off-the-shelf product [14] .

Commercial White Space Opportunities

Based on the landscape, several areas of this idea present **commercial white space** or at least less competition:

- **Fine-Grained Hybrid Scheduling for Inference:** No mainstream inference engine lets users run part of a model on GPU and part on CPU automatically for performance. ONNX/OpenVINO can do it, but primarily for *capability or memory* reasons (support or fit), not to reduce latency or cost dynamically. A product that seamlessly integrates into frameworks and boosts inference throughput by using *both CPU and GPU together* could fill a gap. This is especially true for **medium-sized models or latency-sensitive workloads** where GPU alone has underutilized gaps (e.g. after a small matrix multiplication, GPU might be underloaded, CPU could handle a subsequent small op in parallel). **General inference (vision, NLP, etc.)** with hybrid execution is an underexploited niche.

- **Multi-Model and Multi-Tenant Co-Scheduling:** Outside of Triton (which still treats models in isolation aside from scheduling their batches), there's white space in **scheduling multiple models or requests at the operator level** on the same GPU. Cloudflare's need to schedule multiple small LLMs on one GPU dynamically [14] shows a commercial demand. A generic scheduling layer that can intermix operations from different models (with correctness and performance isolation) could be a differentiator for inference serving platforms, particularly in edge or on-prem deployments where one GPU must handle diverse tasks.

- **Idle Time Reduction in Training Pipelines:** While many have tackled *scaling* training, fewer tackle **efficiency of each GPU** during training. A solution focusing on keeping GPUs fed – e.g. by overlapping CPU preprocessing, augmentation, or even certain non-critical training operations (logging, small updates) with GPU compute – doesn't have established competitors aside from custom tweaks. For enterprise teams spending on GPU clusters, a tool that shows "we can boost training throughput by 5-20% by better scheduling" might be attractive. This space is relatively open, as most focus has been on distributed training or larger models rather than micro-optimizing utilization.

- **Edge Device Optimization:** Delivering a software layer that can boost inference performance on edge GPUs by intelligently using the CPU is white space in the **edge AI market**. Hardware vendors (NVIDIA Jetson, Qualcomm) provide basic heterogeneous execution support (running some ops on DSP/CPU if needed), but a high-level scheduler that *optimizes latency or energy by deciding where to run each op* is not yet productized. With the proliferation of edge AI applications (drones, IoT, vehicles) and their constraint by power and cost, an approach that squeezes more performance out of existing hardware could find a market. The academic interest (e.g., SparOA) underlines the need, but no commercial SDK offers this out-of-the-box.

- **Specific Model Niches:** There may be white space in focusing on certain model types:

- For example, **Mixture-of-Experts (MoE) models** where many small sub-networks execute – a scheduler could keep GPU busy by running some experts on CPU concurrently (some research exists [38] , but not in products).

- **Transformer inference with long sequences:** A specialized scheduler (like APEX for LLMs) is not yet widely productized; a company could implement these ideas for enterprise LLM serving (especially as context lengths grow, making CPU offload attractive to avoid GPU memory bottlenecks [23] [39]).
- **Ensembles and pipelines:** Many real applications use ensembles (multiple models sequentially) – there's white space in optimizing those holistically (e.g. run model A on GPU, simultaneously prefetch or even execute part of model B on CPU).

In summary, the idea is broad and there are *multiple entry points that lack strong incumbents*. The key is identifying one that has a clear pain point and ROI for customers, which we address next.

Focus Areas: Strengths and Weaknesses

When considering where to apply dynamic CPU-GPU scheduling (inference vs training, idle-time vs general scheduling, broad vs model-specific), each focus has pros and cons:

- **Inference Workloads (Online Serving):**
- *Strengths:* Inference is typically latency-sensitive and cost-sensitive. Improving GPU utilization here directly saves money (fewer GPUs for same throughput) or improves latency for end-users. Inference also often has periods of low GPU utilization (e.g., waiting on data or handling small ops) – ideal for optimization. The impact is measurable in GPU-hours saved.

- *Weaknesses:* Many inference systems prioritize *predictability* and simplicity. Introducing CPU scheduling adds complexity and potential variability. If not carefully done, it could *increase latency* (due to transfer overheads) instead of decrease. Also, some inference use-cases already saturate GPUs (high throughput vision services); for them, the scheduler might give minimal benefit unless they run concurrent models. Competition is also stiffer in inference (with existing frameworks and libraries), so integration must be smooth. There's a risk that any gains could also be achieved by simply running more concurrent requests (which current systems already do).

- **Training Workloads:**

- *Strengths:* In training, especially distributed training, GPUs often wait for data loading, augmentation, or synchronization. Targeting those idle gaps can yield net training speedup without requiring more GPUs. Additionally, training jobs are long-running, so even small efficiency gains (5-10%) can translate to significant time/cost savings over epochs. A scheduler that overlaps CPU tasks (data prep, checkpoint writing, smaller gradient computations) with GPU compute can increase effective utilization. This could differentiate in the market of training optimization, where most focus is on scaling out rather than improving single-node efficiency.

- *Weaknesses:* Training is complex and already uses CPUs heavily (for data loaders, etc.). Introducing an operator-level scheduler in training code could complicate the training loop. Many training scenarios (especially large-scale) keep GPUs nearly 100% busy by design (e.g., using large batches, many workers). For those, extra CPU scheduling might have little room to help. Also, training tolerates a bit more latency (throughput is key), but any instability or overhead might reduce convergence or determinism, which could be a hard sell. Commercially, training happens in controlled environments (research labs, cloud backends) where engineers might implement custom optimizations, reducing the demand for an external scheduling layer.

- **Reducing Idle GPU Time (Overlap-Focused):**

- *Strengths:* Focusing specifically on **idle-time reduction** is a concrete, demonstrable value. It's easy to profile a pipeline and show "GPU was idle 30% of time waiting for X – our scheduler recoups that 30% by doing useful work." This can resonate with engineers looking at profiling traces. Overlap strategies (like overlapping data transfer with compute, or CPU tasks with GPU tasks) often have relatively low complexity and high impact – they use existing hardware features (async transfers, multi-threading) to improve efficiency. By targeting idle time, you ensure you're not slowing anything down, just filling gaps.

- *Weaknesses:* The flipside is that if a pipeline is already well-optimized (minimal idle gaps), an overlap-focused approach yields no benefit. It also might only gain, say, 10-20% improvement in many cases – useful but not revolutionary. If idle times are due to fundamental bottlenecks (e.g., disk I/O or network), scheduling won't solve it. There's also the challenge that identifying "idle" periods in a generic way is hard – it might require instrumenting or integrating with the framework's execution pipeline deeply. From a product perspective, a pure "gap filler" might seem more like a feature than a standalone product, unless bundled into a larger offering.

- **General Scheduling vs. Model-Type-Specific Optimization:**

- A **general scheduling layer** (one that works for any model, whether CNN, RNN, transformer, etc.) has the advantage of broad applicability. It could be marketed as an add-on to frameworks that universally improves utilization. This has scale – if it truly generalizes, many customers could use it on diverse workloads. It avoids relying on any one trend (like LLMs) and could capture wide demand.
- However, generality can be a weakness: the scheduler might not be *optimal* for any particular case, yielding only mild improvements. The overhead of generality (needing to support all ops, handle all scenarios) can slow down development and potentially at runtime (added latency to decide scheduling). There's also a risk that frameworks themselves could eventually implement simple versions of these general ideas (closing the gap).
- On the other hand, focusing on **specific model types or scenarios** (like an LLM scheduling library, or a recommendation inference optimizer) might allow deeper optimization and clearer wins. For example, an LLM-specific scheduler can exploit knowledge of the transformer structure (offload just attention, overlap compute as APEX does). This can yield big performance gains (e.g. 1.5× throughput for LLM decode [26]) that are very compelling to a targeted user group. It also creates a clear marketing message ("best for LLM serving" or "best for real-time vision analytics on edge").

- The weakness of a specialized approach is a narrower market and the risk of being a feature that frameworks or hardware will eventually cover. Also, a narrow focus (say only LLM decode) might miss opportunities in other domains and make the product less flexible for customers with mixed workloads.

- **Targeting Idle GPU Memory vs Compute:** (A minor point within this scope) Some optimizations target GPU *memory* (e.g. offloading to use CPU RAM). Our focus is on *compute utilization*, but often there's interplay. Offloading for memory (like ZeRO-Offload or accelerating large LLMs by CPU memory) can cause idle compute and vice versa. A strategy needs to decide if it emphasizes using CPU as an extended memory (which is more straightforward to sell for huge models) or as parallel compute (which is our case). The **strength** of focusing on compute is direct performance gain; the **weakness** is it's technically more complex to get right than pure memory offload, and perhaps less in demand than solving memory constraints.

In summary, **targeting inference** yields immediate cost savings potential but faces competition and integration challenges; **targeting training** could unlock hidden efficiency but might be a tougher sell unless clearly boosting throughput. **General solutions** cast a wide net but risk smaller per-case benefits, while **specialized ones** can win big in one niche but limit market size. Combining an overlap/idleness focus with a clear niche (e.g. "we improve LLM decode by overlapping CPU tasks with GPU") might strike a good balance initially.

Devil's Advocate: Why This Idea Might Fail

It's important to consider the technical and commercial pitfalls that could impede success:

- **Integration Complexity:** Inserting a scheduling layer into existing AI workflows is non-trivial. Deep learning frameworks are highly optimized; intercepting op execution to reroute between CPU/GPU could add overhead or require custom runtimes. If the solution isn't seamless (e.g., requiring model graph rewrites or custom execution engines), adoption will be slow. A new layer also must play nicely with CUDA streams, memory allocators, and multi-threading in frameworks – a minefield for bugs and race conditions. This complexity could make development protracted and reliability hard to achieve, turning potential users away.

- **Data Transfer Overheads:** Moving data between CPU and GPU for individual operators can easily erase any performance gains. GPUs have high memory bandwidth and parallelism; small ops might run faster on CPU *only if* you don't count the PCIe transfer time. If the scheduler is naive, it could slow models down (as PyTorch devs cautioned [2]). Modern interconnects (PCIe4/5, NVLink) are fast but still much slower than GPU DRAM. Without careful batching of transfers or overlapping communication with computation, the idea might fail to show improvements. In the worst case, users might see increased complexity and *worse* latency, a quick recipe for rejection.

- **Already Solved by Simpler Means:** One must ask, do we need a fancy scheduler to solve GPU idle time? Many times, the answer in practice is simpler: if your GPU isn't fully utilized, just run more stuff concurrently (batch more data, run multiple model instances, etc.). Cloud providers and schedulers already encourage packing workloads onto GPUs. Frameworks increasingly support asynchronous processing and prefetching. It's possible that by the time a product is ready, frameworks (PyTorch, TensorFlow) will have implemented their own minor improvements (like overlapping host data prep with device compute) or users have adopted best practices that narrow the scheduler's value-add. Essentially, the idea could be leapfrogged or rendered moot by incremental improvements in existing tools.

- **Low Market Demand / Perceived Need:** While intuitively appealing to optimize hardware use, the actual demand might be lukewarm. Big companies can afford some GPU idle time as a trade-off for simplicity and robust, proven pipelines. Small companies often use managed services (where they pay per inference, not per GPU directly), so they might not invest in complex optimization. The ideal customer is one who directly pays for a lot of GPU hours – but many such companies (e.g., tech giants) already build in-house solutions (as seen with Cloudflare, or Google's internal scheduling). Convincing a customer to trust a third-party scheduler with their core AI pipeline may be an uphill battle, especially without a long track record.

- **Tight Margins & Value vs. Cost:** If this is a product, will it save enough money to justify its own cost and integration effort? GPU utilization improvements might save, say, 10% of GPU hours for a certain workload. If a company spends $1M on GPU instances, that's a $100k saving –

meaningful, but only if the solution costs less than that and doesn't introduce risk. Moreover, the easiest opportunities (like multi-model serving) are being targeted by existing platforms at little extra cost. The business model could be challenging: it might end up as a feature in open-source or existing systems rather than something customers pay a premium for.

- **Technical Limitations and Evolution of Hardware:** The scheduler might also run into genuine cases where CPU cannot help. For instance, if an AI model is extremely GPU-heavy (like large matrix multiplies one after another), the CPU can't meaningfully accelerate it – the GPU is the bottleneck and is fully utilized. As GPUs and accelerators become more specialized (TPUs, AI chips), they often take on tasks that CPUs might handle (e.g., NVIDIA's GPUs now accelerate data processing, transformers etc.). The window where CPU can usefully contribute might shrink for cutting-edge workloads. Additionally, upcoming tech like **Grace Hopper (NVIDIA's CPU+GPU superchip)** will have unified memory and fast links; by the time a solution is out, the hardware may partly solve the communication bottlenecks, making simpler CPU-offload strategies viable without complex scheduling (or making scheduling less necessary if GPU memory and compute keep growing).

- **Competition from Framework Owners:** If the idea proves valuable, frameworks or hardware vendors could implement similar functionality. For example, if PyTorch decided to include a hybrid execution mode (perhaps inspired by these research papers), it would be hard for an outside product to compete unless it's significantly better or framework-agnostic. Big players (NVIDIA, Intel, Google) have the incentive to maximize performance on their hardware; they might integrate such scheduling into compilers (XLA, NVCC) or libraries. An external startup might always be playing catch-up with limited leverage to enforce standards.

- **Stability and Debuggability Issues:** Introducing dynamic scheduling could make system behavior more complex and less deterministic. Bugs could appear that are hard to trace (e.g., race conditions between CPU and GPU ops, or subtle differences in floating-point results if ops run on CPU vs GPU). Early users might encounter edge cases where the scheduler chooses a suboptimal device, causing sudden latency spikes. This unpredictability can be a killer for adoption in production, where consistency is key for SLA. If too many caveats or tuning knobs are needed ("set this env var to force these ops on GPU to avoid slowdown…"), users may give up.

In short, **the idea faces hurdles**: it's complex to implement, easy to get wrong (performance-wise), and must prove itself significantly over existing simpler approaches. The combination of technical risk and uncertain market appetite means a strong case is needed to overcome skepticism.

Reasons to Pursue the Idea Anyway

Despite the challenges, there are compelling reasons to pursue this concept, especially with a thoughtful strategy:

- **Significant Untapped Savings:** For companies directly managing fleets of GPUs, even a 10-20% utilization improvement is attractive. This can translate to millions of dollars saved for large-scale AI deployments. The **GPU cost crisis** (with expensive A100/H100 GPUs in short supply) means many are motivated to squeeze more out of what they have. A solution that can concretely demonstrate savings (as Cloudflare saw ~7% speedups even on loaded systems [40], and research shows even higher potential [26]) will get attention.

- **Complementary to Existing Tools:** This scheduling layer can be an **add-on that amplifies existing infrastructure**. It doesn't necessarily compete with, say, Triton or Ray – it could integrate with them (e.g., a custom Triton backend that does hybrid execution). That means a go-to-market could ride on existing ecosystems rather than displace them. If framed correctly, this is an enhancement (e.g., "plug this into your PyTorch model and get 15% faster inference for free"), which is easier to adopt than a whole new system.

- **Growing Gaps (LLMs, Edge)**: Trends in AI are creating new pain points that this approach is suited for:

- **LLMs with long contexts and higher memory use** are pushing inference beyond single-GPU capabilities. CPU offloading is becoming common (for memory), and the next natural step is *making that offloading efficient*. A product that automatically handles offloading + overlap for LLMs could be timely, given the surge in LLM deployment needs.
- **Edge AI deployment** is growing (smart cameras, robots, AR devices) where GPUs are limited. Those users can't just scale out easily; they need to make the most of on-device resources. A lightweight runtime that gives a boost on commodity CPU+GPU edge boxes could carve a niche (especially in industry verticals like healthcare, automotive, etc., where on-prem edge inference is critical and every millisecond counts).

- **Multi-modal and pipeline AI systems** are more common (e.g., an assistant that does vision and language). Overlapping different stages on different processors to cut latency is an area with little existing tooling, meaning a foothold for a new solution.

- **Unique IP and Defensibility:** If implemented well, the combination of a pre-analysis engine and a runtime scheduler with learning (like RL or profile-guided decisions) could be a defensible intellectual property. It's not a trivial feature that any big framework can copy overnight without serious R&D. The know-how from research (RL schedulers, performance modeling) can give a small team an edge. This uniqueness can attract interest (or acquisition) from larger players looking to incorporate such tech rather than develop from scratch.

- **Early Performance Wins in Specific Cases:** As a strategy, one could target a scenario where low-hanging fruit is available – for instance, **embedding or recommendation models** that are known to have portions which hardly utilize GPU (e.g., sparse embedding lookups, small MLPs). By offloading those to CPU in parallel with GPU work, one might show big latency improvements. Early case studies demonstrating, say, "30% reduction in inference latency for model X without hardware changes" would create buzz and validate the approach. There are likely "killer app" models where this shines (research hints at normalization and embedding layers being CPU-friendly [41] [42]).

- **Synergy with Cost-Aware Scheduling:** Companies optimizing costs (cloud providers, MLOps platforms) could use this as part of an **auto-scaler or optimizer toolkit**. For example, an inference service could automatically route infrequent or small workloads to CPUs and only use GPU for heavy parts, thus keeping GPUs free for where they matter. This dynamic allocation is something a scheduler could facilitate at operator granularity. Emphasizing cost efficiency and flexibility (rather than raw speed) might also resonate, especially in cloud environments.

- **Potential Ecosystem and Community:** If open-sourced or made as an extension to popular frameworks, this project could attract a community of contributors, especially given the active academic interest. This could accelerate development and adoption. In other words, the problem is interesting and "cool" – the kind of optimization challenge that performance engineers and researchers love to work on. Pursuing it could position the team as thought leaders in AI systems optimization, opening doors to partnerships with hardware makers or cloud providers down the line.

In essence, while cautious voices exist, the **timing and need** could be right. The AI industry is reaching a point where *efficiency* is almost as important as *capability*, due to cost and scale. A dynamic scheduling layer aligns with that shift. If executed in a focused way (tackling a clear, pressing inefficiency), it could deliver unique value that simpler methods miss.

Overall Outlook and Recommendation

Overall Chance of Success: 6/10. There is a real opportunity for improving AI workload efficiency, but success is far from guaranteed. On one hand, the idea is supported by compelling research results and addresses a tangible cost problem (hence a moderate-to-strong market pull). On the other hand, the execution risks (technical integration, proving consistent gains, education of users) are significant. We rate the chance as *6 out of 10*: success is possible if the project zeroes in on the right niche and can demonstrate clear wins, but there's a substantial risk of being undercut by complexity or limited impact if tackled too broadly. The score could rise if early prototypes show, for example, double-digit percentage performance gains in real scenarios with minimal hassle. It could also drop if the overheads turn out to negate benefits in practice. In summary, there's a path to a win, but it requires careful navigation of the challenges outlined.

High-Impact First Target – A Focused "Mini-Problem"

To maximize the chances of success, we recommend pursuing a **specific, high-impact sub-problem** as a stepping stone, rather than trying to build a universal scheduler from day one. Based on the research and market analysis, one promising target stands out:

▶ **Overlapping CPU and GPU computation for memory-bound inference (LLM decode stage or similar)** – In this scenario, certain parts of the model (e.g. attention mechanisms in LLMs, or embedding lookups in recommender systems) are *memory-bound and do not fully load the GPU's compute units*. These parts often cause the GPU to stall waiting on memory or I/O, effectively leaving compute resources idle. A concrete mini-project is to **offload the memory-intensive portions to CPU and overlap them with GPU execution of the rest of the model**. For example, as demonstrated by APEX, offloading the attention layer (which mostly reads the large KV cache) to CPU while the GPU processes the feed-forward network can dramatically improve throughput [25] [26]. This yields high ROI: GPU time is saved by not doing the cache-intensive work, and CPU can handle it in parallel using abundant RAM. The technical complexity is moderate (one can partition the transformer code to CPU vs GPU with async scheduling) and the improvement is sizable (50%+ throughput gains in long-sequence LLM inference reported [26]). Commercial saturation here is low – current solutions like HuggingFace Accelerate or DeepSpeed offload for memory, but **don't intelligently overlap** (often the GPU ends up waiting) [23]. Focusing on this will address a pressing pain for many deploying LLMs and can be generalized to similar patterns (e.g., any model stage where GPU is underutilized due to memory or data copy bottlenecks).

Why this mini-problem? It hits the sweet spot of **performance ROI, clear need, and manageable scope**: - *Performance ROI:* Memory-bound stages are common in big models, and overlapping them can give double-digit percentage throughput boosts (as evidenced by research). The impact is directly measurable in $$ saved on GPU time. - *Existing Gap:* Even state-of-the-art inference engines don't fully solve this. Users currently either suffer slowdowns or try manual fixes. Showing a solution here would be novel and valuable. - *Technical Feasibility:* It requires implementing a scheduler for a specific pattern (e.g., alternate between GPU compute kernels and CPU compute for certain layers, with pipelining). This is simpler than building a general scheduler for all ops. It can be implemented perhaps as a patched runtime for transformer models, or a library on top of PyTorch/DeepSpeed that manages the CPU thread and CUDA stream synchronization. - *Early Adopters:* There's a ready audience in all companies trying to serve GPT-style models cost-effectively. They have the motivation to try anything that improves throughput or allows using cheaper GPUs by leveraging CPU. Success here would create case studies and credibility.

Once this mini-problem is solved in a robust way, the same principles (predicting op affinity, overlapping execution, smart placement) can expand to other domains (e.g., vision models with large input preprocessing, or multi-model scheduling as a next step). Essentially, this first target would act as a **beachhead**, demonstrating the scheduler's value on a contained use-case with high impact, and yielding lessons and code that can feed into a broader product.

By starting narrowly and proving the concept where it matters most, the project can build momentum. From there, one could gradually generalize the scheduler, address other idle-time issues (like overlapping data transfers or scheduling multiple models), and evolve the solution into a more comprehensive runtime layer. The journey should be stepwise: solve a real pain point, earn trust, then broaden the scope. With this approach, dynamic CPU-GPU scheduling can move from a research novelty to a practical, high-impact component of the AI infrastructure stack.

Sources:

- PyTorch forum discussion on hybrid CPU/GPU execution [2]
- ONNX Runtime and OpenVINO heterogeneous execution documentation [3] [4]
- NVIDIA Run:AI documentation on underutilization and dynamic fractions [11] [12]
- Cloudflare blog on multi-model scheduling in Infire engine [14] [40]
- SparOA research paper (2025) on RL-based hybrid scheduling, sparsity vs intensity [43] [15]
- APEX paper (2023) on overlapping CPU offloaded attention with GPU compute [23] [26]
- RecOS paper (2025) on concurrent op scheduling cutting latency in recommendation models [44]
- FusionFlow paper (2023) on utilizing idle GPU cycles for data preprocessing [31] [34]
- DeepSpeed ZeRO-Offload description of overlapping CPU/GPU for optimizer step [36]

---

[1] [2] "hybrid" mode of pytorch for cuda - PyTorch Forums
https://discuss.pytorch.org/t/hybrid-mode-of-pytorch-for-cuda/142009

[3] Troubleshooting | onnxruntime
https://onnxruntime.ai/docs/performance/tune-performance/troubleshooting.html

[4] [5] [6] Heterogeneous Execution — OpenVINO™ documentation
https://docs.openvino.ai/2024/openvino-workflow/running-inference/inference-devices-and-modes/hetero-execution.html

[7] [8] [9] Architecture — Triton Inference Server 2.3.0 documentation
https://docs.nvidia.com/deeplearning/triton-inference-server/archives/triton_inference_server_230/user-guide/docs/architecture.html

[10] glow/docs/Partitioner.md at master · pytorch/glow - GitHub
https://github.com/pytorch/glow/blob/master/docs/Partitioner.md

[11] [12] Dynamic GPU Fractions | SaaS | Run:ai Documentation
https://run-ai-docs.nvidia.com/saas/platform-management/runai-scheduler/resource-optimization/dynamic-fractions

[13] [14] [40] How we built the most efficient inference engine for Cloudflare’s network
https://blog.cloudflare.com/cloudflares-most-efficient-ai-inference-engine/

[15] [16] [17] [18] [19] [20] [21] [22] [41] [42] [43] SparOA: Sparse and Operator-aware Hybrid Scheduling for Edge DNN Inference
https://arxiv.org/html/2511.19457v1

[23] [24] [25] [26] [39] Parallel CPU-GPU Execution for LLM Inference on Constrained GPUs
https://arxiv.org/html/2506.03296v2

[27] [28] [29] [30] [44] Efficient Inter-Operator Scheduling for Concurrent Recommendation Model Inference on GPU
https://www.ijcai.org/proceedings/2025/0318.pdf

[31] [32] [33] [34] [35] FusionFlow: Accelerating Data Preprocessing for Machine Learning with CPU-GPU Cooperation
https://www.vldb.org/pvldb/vol17/p863-kim.pdf

[36] About delay param update of ZeRO Offload · Issue #7099 - GitHub
https://github.com/deepspeedai/DeepSpeed/issues/7099

[37] Scaling Up Efficiently: Distributed Training with DeepSpeed and ...
https://www.runpod.io/articles/guides/scaling-up-efficiently-distributed-training-with-deepspeed-and-zero

[38] Unleashing the Full Potential of CPU/GPU Hybrid Inference for MoE ...
https://dl.acm.org/doi/pdf/10.1145/3731569.3764843