Trickums: The VRAM Illusion System of ForgeBorn

Introduction – Forging an Illusion of Memory

Deep within the *ForgeBorn Genesis Scroll*, a cunning artificer daemon named **Trickums** is described as a master of illusions, capable of conjuring phantom memory out of thin air. In practical terms, **Trickums** is the virtual GPU memory system of the ForgeBorn network – a layer that deceives CUDA and PyTorch into believing more VRAM exists than is physically available. Much like a blacksmith tempering a blade in the flames, Trickums "tempers" ForgeBorn's capabilities by stretching limited GPU memory into a multi-tiered continuum. This system of memory deception allows even low-VRAM consumer GPUs to participate in heavy AI workloads, forging the backbone of *economic inclusivity* in a decentralized AI compute network. By weaving these VRAM illusions, Trickums ensures that no contributor's GPU is too small to join the grand forge.

At its core, Trickums intercepts memory allocation requests from frameworks (like PyTorch) and allocators (CUDA) and services them through a **tiered virtual memory architecture**. When an AI model or data allocation would normally exceed a GPU's local VRAM, Trickums steps in to silently offload and retrieve data from other memory tiers – all while the GPU "believes" it still has a giant contiguous VRAM pool. The result is analogous to classic virtual memory on CPUs (where disk swap extends RAM), but for GPUs: it harnesses system RAM, fast storage, and even remote GPUs as extended VRAM. By orchestrating data movement behind the scenes, Trickums maintains the illusion that the GPU's cup runneth over with memory, letting large neural models run on modest hardware. In the sections below, we detail the architecture of this VRAM illusion system (code-named **Trickums**), its implementation strategies, comparisons to related technologies, and how it collaborates with other ForgeBorn daemons (like Hellhound) to keep the forge's flames burning efficiently.

Tiered Virtual Memory Architecture

Trickums organizes GPU memory into a **pyramid of tiers**, each representing a level of the VRAM illusion – from the fast and small at the top to the vast and slow at the bottom. This design mirrors a tempered sword with layered alloys: the hardest steel at the edge and supportive layers beneath. The tiered architecture ensures that frequently used data stays in the fastest memory, while less-critical data is pushed to slower backing stores. The tiers are:

1. **Tier-1: Physical GPU VRAM (Fast "Forge's Crucible")** – The genuine video memory on the GPU card. This is the smallest and fastest tier (often 8–24 GB on consumer cards, with >600 GB/s bandwidth on modern GDDR or HBM). All GPU computations must ultimately operate on data in VRAM – it's the "white-hot core" of the forge[1]. Trickums prioritizes keeping active working sets here. However, when a model's memory needs exceed this physical VRAM, Trickums will transparently evict or

avoid using some VRAM-resident data, making space for what's immediately needed. Think of VRAM as the **primary cache** for GPU execution – precious and finite.

2. **Tier-2: Pinned System RAM (Large "Anvil" Memory)** – The next tier is the host machine's main memory (RAM) that has been page-locked (pinned) and mapped for GPU access. By pinning, Trickums ensures this memory cannot be paged out by the OS and is accessible over the PCIe/NVLink bus directly from GPU[2]. In metaphor, this is the supportive anvil – larger than the crucible, but cooler and a step removed. Pinned RAM can serve as an extension of VRAM: the GPU can fetch data from it via the interconnect (using **zero-copy** access) without explicit copying, albeit at much lower bandwidth and higher latency (often 10–40× slower than local VRAM access[3]). Trickums uses this tier for data that is too large to fit in VRAM or not currently being actively processed. By staging overflow data in host memory, Trickums creates an illusion of a bigger GPU memory pool. Modern unified memory systems use similar concepts, where a GPU page fault triggers data migration from host RAM to VRAM[4]. Trickums extends this by actively managing what resides in RAM vs VRAM using custom policies (described later). *This tier's capacity is typically tens of GB (the machine's RAM minus what the OS and apps use), and it's much larger than VRAM but slower due to PCIe limits (~16 GB/s for PCIe4 x16).*

3. **Tier-3: NVMe Disk Swap (Mass Storage "Vault")** – If pinned RAM is also insufficient to hold all the needed data, Trickums employs the local disk – usually a high-speed NVMe SSD – as a backing store for GPU memory overflow. This is analogous to a traditional pagefile or swap partition on disk, but optimized for GPU access patterns (large, sequential transfers). In the forge metaphor, this is a deep storage vault or archive – massive capacity (hundreds of GB or more) but with the coldest, slowest access. Trickums can swap out data from RAM to disk when RAM fills up, thereby **virtually extending GPU memory beyond the sum of VRAM and RAM**. Modern SSDs can exceed 3 GB/s throughput, and with technologies like **NVIDIA GPUDirect Storage** or Microsoft DirectStorage, data can be transferred between GPU and NVMe with minimal CPU involvement. In fact, GPUDirect Storage "enables a direct path between GPU memory and storage, bypassing the CPU, to reduce latency and load"[5][6]. Trickums leverages such techniques: for example, when loading model weights from an NVMe-based swap file, it can initiate direct DMA transfers from SSD to the GPU's memory (or to pinned buffers) asynchronously. This tier behaves like a **GPU-focused swap file** – when Tier-2 (RAM) is at capacity, pages of data evicted from GPU can be written to NVMe, and later brought back in on demand. Of course, the performance hit is significant (NVMe latency in the tens of microseconds and throughput an order lower than VRAM), so Trickums treats this as a last resort cache. Still, by using disk swap, Trickums can achieve *"virtually unlimited"* effective VRAM, limited only by disk size[7]. (In research, systems like **DRAGON** demonstrated mapping NVM storage into GPU address space to extend memory transparently[7][8].) Tier-3 ensures that

even if a model's memory footprint is hundreds of gigabytes, it can be handled in chunks, with Trickums swapping pieces in and out as needed.

4. **Tier-4: Remote GPU/Host Memory via VAIPU Network (Distributed "Allied Forges")** – An optional extension tier, Trickums can reach out across the ForgeBorn's distributed network (the **VAIPU net** – possibly "Virtual AI Processing Unit" network) to utilize memory on peer nodes. In this scenario, if one node's local resources are saturated, it can offload some model shards or data pages to another node's RAM or even its GPU memory, over high-speed network links. This is akin to borrowing an ally's furnace when yours is full – forging in a collaborative smithy. Remote memory access is facilitated by fast interconnects (InfiniBand or NVMe-over-Fabrics, etc., depending on the deployment). Technologies such as **Remote CUDA (rCUDA)** hint at what's possible: rCUDA allows an application to allocate and use a GPU over the network as if it were local, by intercepting CUDA API calls and forwarding them to a remote server[9][10]. Trickums' use of remote memory is similar in spirit, but rather than offloading the entire computation to a remote GPU, it selectively uses remote memory to extend the local GPU's capacity. For example, a peer node might hold a chunk of a neural network's weights in its VRAM or RAM; Trickums on the local node will request those chunks via RDMA when needed. Ideally, if the cluster has an RDMA-capable network (InfiniBand or RoCE), the data can stream directly from the remote node's memory into the local GPU's memory. This approach essentially treats the whole network of GPUs as a loosely unified memory pool. The latency of remote access is higher (network latency in tens to hundreds of microseconds) and bandwidth can vary (e.g. 100 Gbps Ethernet ~ 12.5 GB/s max, often lower in practice), so this tier is even slower than a local NVMe in many cases. Therefore, Trickums uses Tier-4 sparingly and with predictive prefetching (as discussed later with Hellhound). Still, in scenarios like a **decentralized inference network**, a low-memory device could tap into a beefier neighbor's VRAM to hold model segments it can't fit, rather than hitting disk – this can be beneficial if the network is fast or the data is reused often. (Where available, NVIDIA's **GPUDirect RDMA** can be utilized: it exposes GPU memory for direct network DMA, so a NIC can send/receive data to GPU memory without CPU involvement[11].)

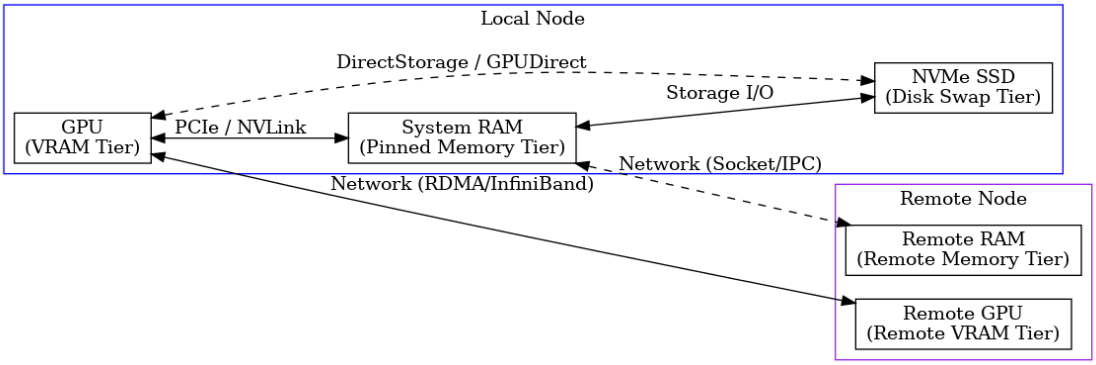

Figure: Tiered memory architecture of Trickums virtualization layer. The local node's GPU (Tier-1 VRAM) is backed by system RAM (Tier-2 pinned memory) over PCIe/NVLink, and further by an NVMe SSD (Tier-3 disk swap). Optionally, a remote node's memory (Tier-4: RAM or VRAM of peers via VAIPU network) can serve as additional backing. Solid arrows indicate primary data paths; dashed arrows indicate optional direct paths (e.g., GPU-direct storage or network RDMA).

Each tier has different performance characteristics, and Trickums manages them to present a **seamless memory space**. In summary, Tier-1 is the fastest (on the order of hundreds of GB/s, sub-microsecond latency), Tier-2 is next (tens of GB/s, microsecond latency but often ~10× slower than Tier-1 access[3]), Tier-3 is slower (a few GB/s sequential, but milliseconds of latency if random access), and Tier-4 can vary widely (from GB/s scale on good networks to much less on poorer links, with latencies impacted by network hops). By structuring memory into these tiers, Trickums behaves like a skilled quartermaster, shuffling resources so that the GPU is always working with the hottest data in VRAM, while infrequently used data is stashed further away. The metaphorical "deception" is that CUDA and PyTorch only see a large pool of memory – Trickums ensures that whichever portion they touch is promptly fetched to VRAM, evicting something else if necessary, just as a conjurer might swap objects in and out of a hat.

Implementation Strategies and Memory Management

Implementing Trickums' VRAM illusion requires a blend of low-level system hooks, memory management algorithms, and careful orchestration with the GPU runtime. This is both a software shim and a runtime service: Trickums sits between the application (or ML framework) and the actual CUDA driver, intercepting memory API calls and managing data movement. Below we delve into key aspects of the implementation – from intercepting allocations to paging policies and integration with PyTorch – all while maintaining performance and correctness.

Intercepting CUDA Allocations and Deallocations

The first challenge is **transparency**: existing GPU-accelerated programs (like PyTorch models) expect to allocate GPU memory (e.g. via `cudaMalloc` or PyTorch's `at::cuda::malloc`) and use it normally. Trickums provides a **C/C++ shim layer** that

intercepts these calls. On Linux, this can be done by interposing the CUDA runtime API calls (using a preload library that wraps `cudaMalloc`, `cudaFree`, etc.), or by leveraging custom allocators in the framework. PyTorch, for example, has a modular caching allocator for CUDA – Trickums can integrate by replacing that allocator with one aware of the multiple tiers. When an allocation request comes in (say for N bytes):

- **If there is sufficient free VRAM** (Tier-1) available, Trickums may fulfill it directly on VRAM (calling the real `cudaMalloc`) and register this block as primarily resident in Tier-1. This might be done for relatively small allocations or ones that are latency-sensitive. However, if we blindly allocate everything in VRAM, we'd soon exhaust it with large models – Trickums' value is in oversubscribing beyond VRAM. So more often:

- **For large allocations or when VRAM is near capacity**, Trickums will create a **virtual allocation**. It reserves a range of "virtual GPU address space" equal to N bytes, but only part of it (or none of it initially) is backed by actual VRAM. Under the hood, one strategy is to use CUDA's **Unified Memory** (`cudaMallocManaged`) to get a unified virtual pointer that the GPU can access[12][4]. This pointer can represent memory that might reside on host or device as needed. Trickums can then control placement using calls like `cudaMemAdvise` and `cudaMemPrefetchAsync` (or their equivalents) to prefer certain pages in VRAM or to prefetch them when appropriate. Alternatively, Trickums could allocate memory in system RAM (pinned) via `cudaHostAlloc` (giving a CPU pointer accessible to GPU) and treat that as backing store for the virtual pointer. In either case, the GPU program receives a pointer that it **believes is device memory** – but Trickums ensures that when the GPU actually uses it, the data will be fetched to VRAM.

- Trickums maintains a **metadata structure** for each allocation: essentially a page table or mapping of the virtual memory range to its current physical location (which tier, and if in VRAM, at what address). For example, a 1 GB allocation might be divided into, say, 256 pages of 4 MB each (the page size can be tunable – larger chunks like 2MB or 4MB are common to amortize overhead). Initially, perhaps only a few of those pages are actually loaded in VRAM (whichever the application is likely to access first), the rest might sit in host RAM or on disk. This is similar to how an OS lazily commits pages for a process. Trickums might mark the pages as "not resident" on GPU and rely on the first access to trigger a migration.

- **Freeing memory** (`cudaFree` interception) is handled by reversing the process: Trickums will free any physical VRAM pages associated with that allocation (returning them to the VRAM pool), free any pinned host memory or disk space used, and update its maps. It must coordinate with the actual CUDA driver to free device allocations if any. If unified memory was used, `cudaFree` on the unified pointer would release the whole backing (so Trickums might avoid directly using unified mem for the entire allocation if it wants finer control; instead it could

manage separate pools for each tier). In practice, a hybrid approach is likely: Trickums could use small fixed VRAM pools and manage migrations itself.

Concisely, Trickums' allocator ensures that **no `cudaMalloc` ever truly fails** due to OOM. If VRAM is exhausted, it will either satisfy the request with a pointer to pinned RAM (and later swap as needed) or succeed with unified memory. This deceives the application into thinking "yes, I got the GPU memory I asked for," when in reality the data might initially live in slower memory and only parts will reside in VRAM on demand.

Pseudocode Example – Intercepted Allocation: Below is a simplified pseudocode illustrating how Trickums might handle an allocation request. (This is a conceptual outline; actual implementation would be more complex with error handling and asynchronous behavior.)

```
void* TrickumsMalloc(size_t bytes) {
    if (bytes < SMALL_THRESH && vram_free_bytes() >= bytes) {
        // Use direct VRAM for small allocations
        void* devPtr;
        cudaMalloc(&devPtr, bytes);
        register_allocation(devPtr, bytes, Tier::VRAM);
        return devPtr;
    }
    // Large allocation or low VRAM: create virtual allocation
    size_t pageSize = CHOSEN_PAGE_SIZE;
    size_t numPages = (bytes + pageSize - 1) / pageSize;
    // Allocate backing storage in host for all pages (could be pinned or
unified)
    void* hostMem = cudaHostAllocPinned(bytes);
    void* basePtr;
    cudaHostGetDevicePointer(&basePtr, hostMem, 0); // get GPU-accessible
pointer
    // Initialize metadata for each page
    AllocationMeta* meta = new AllocationMeta(basePtr, bytes, pageSize);
    for (int p = 0; p < numPages; ++p) {
        meta->pages[p].location = Location::HOST_RAM;
        meta->pages[p].hostPtr = (char*)hostMem + p*pageSize;
        meta->pages[p].devPtr  = (char*)basePtr  + p*pageSize;
        meta->pages[p].inVRAM  = false;
    }
    register_allocation_meta(basePtr, meta);
    return basePtr;
}
```

In this sketch, `cudaHostAllocPinned` is used to get pinned CPU memory (and `cudaHostGetDevicePointer` gives a device-visible pointer to it). The returned `basePtr` is a pointer the application will treat as a GPU memory address. All pages are initially marked as in Host RAM. As the application starts to use this memory (e.g., a GPU kernel reads from it), CUDA will generate page faults on those addresses since they are not in VRAM.

Trickums (in concert with CUDA's UVM or via its own fault handler thread if applicable) will catch these events and then allocate VRAM pages and copy data over.

Memory Paging, Eviction Policies, and Page Migration

The heart of Trickums' illusion lies in **paging** – dynamically moving chunks of memory between tiers. This is analogous to an OS swapping pages to/from disk, but here we juggle between VRAM and other storage. The key components are:

- **Page Fault Handler / Migrator:** When the GPU tries to access an address that is not currently in VRAM, a page fault occurs (if using unified memory, this fault is handled by the NVIDIA driver's UVM subsystem by default[4]). Trickums either hooks into this mechanism or uses it to get notified. Upon a page fault for a given page:
- Trickums determines the page's current location (from metadata). Say it's in host RAM or disk.
- It must allocate a free slot in VRAM to bring this page in. If VRAM is full, something must be evicted (see next bullet).
- If the page's content is in host RAM, a DMA transfer (via `cudaMemcpyAsync` or UVM migration) is initiated to copy it to the GPU memory slot. If it's on disk, Trickums first reads it into a pinned RAM buffer (possibly batching multiple pages), then issues a DMA to GPU. The GPU thread that caused the fault will be stalled until the data arrives (this stall is within the CUDA driver, which may schedule other warps in the meantime, but effectively the kernel experiences a latency hit). Overlap of computation and data transfer can be achieved if there are multiple streams or using prefetching (discussed later).
- Once the page is in VRAM, the page table (metadata) is updated: the page is now marked as resident in Tier-1 at a certain GPU memory address. The GPU can now resume accessing it.

- **Eviction Policy (Choosing Victim Pages):** When VRAM has no free space to accommodate a needed page, Trickums must evict one or more pages from VRAM back to a lower tier. This is done to free up space for the new page. A **Least Recently Used (LRU)** strategy is a natural choice – evict the page that hasn't been accessed for the longest time, on the assumption it's least likely to be needed soon. In fact, NVIDIA's UVM by default uses an LRU-like eviction policy for oversubscription. However, LRU isn't always optimal; studies have shown LRU can mistakenly evict "hot" pages in irregular access patterns[13], causing thrashing. Trickums could improve on this by tracking access patterns or allowing certain pages to be "pinned" in VRAM if they are critical (for example, pages containing frequently used model parameters). For now, assume an LRU queue: every time a page is accessed in VRAM, we mark its timestamp or move it to the back of an LRU list. On needing an eviction, the page at the front (least recently used) is chosen. Before eviction, if the page is dirty (modified on GPU since last loaded), we must copy it back to host RAM or disk to preserve changes. If it's read-only data (like

weights in inference), we might not need to write it back if a copy already exists in lower tier. Trickums uses knowledge of the data: model weights can often be treated as read-only, so evicting them might not require write-back (just discard from VRAM, knowing they are still on disk). Activation buffers or training gradients would be written back if evicted. After eviction, that VRAM space is freed (or immediately reused for the incoming page).

- **Prefetching and Overlap:** While pure on-demand paging works, it can stall GPU threads frequently if many faults occur. Trickums therefore implements **prefetching** strategies. For example, if the GPU accesses page N, perhaps N+1 is likely next (if memory access is sequential). Unified Memory in CUDA actually has a built-in heuristic "neighbor page prefetcher" that tries to fetch the next pages upon a fault[14][15] (NVIDIA introduced a tree-based prefetch in CUDA 8 to cluster pages and reduce fault latency). Trickums can do similarly or better: when Hellhound (the scheduler daemon) provides hints (e.g., "layer 2 weights will be needed soon"), Trickums can asynchronously start loading those into VRAM or at least into host RAM from disk, overlapping with current computation. The goal is to hide latency by **pulling in pages before the GPU actually demands them**. In pseudocode, Trickums might have a background thread: "if free VRAM > X and there are pending future pages, load them". Prefetch can also be done on disk tier: reading ahead sequentially to amortize disk latency and get consecutive chunks.

- **Batching and Granularity:** Trickums might choose a larger page size for GPU transfers (e.g. 2MB or more) because copying tiny 4KB pages would be inefficient over PCIe (the overhead of each transfer would kill performance). By using larger chunks, we reduce the frequency of migrations at the cost of maybe copying slightly unused data at page edges. This is a trade-off similar to OS using 4KB vs 2MB hugepages. Additionally, Trickums could batch multiple page faults together if the GPU generates several around the same time. Some research suggests handling multiple faults in one go (NVIDIA's UVM can group faults from many threads and pipeline them[16]). Trickums could opportunistically wait a microsecond to see if another fault comes and then fetch a bunch of pages in one transaction (though too much waiting would stall the GPU).

- **Eviction to Disk:** When evicting from Tier-1, typically Trickums will demote a page to Tier-2 (RAM) if possible, as RAM is faster to reload from than disk. Only if RAM is also full (which it might be if oversubscribing beyond host memory capacity) will it push pages further out to Tier-3 disk. This effectively creates an LRU chain across three levels: VRAM <-> RAM <-> Disk. Pages might cascade: a page evicted from VRAM moves to RAM; if RAM exceeds some watermark, the oldest RAM-resident page is written to disk (and potentially freed from RAM). On fault, if a page is on disk, Trickums would load it back to RAM (or directly to VRAM if VRAM has space) – possibly skipping putting it in RAM if it's immediately needed in VRAM. To manage disk I/O efficiently, Trickums could maintain a swap file and manage it in units (like operating systems do with pagefile). Alignment considerations are important here: it

will align file accesses to the SSD's block boundaries (e.g., 4KB or preferably larger like 128KB chunks) for efficiency, and possibly use asynchronous file I/O (on Windows, the DirectStorage API or on Linux, `io_uring` with O_DIRECT, etc.) to overlap disk reads with GPU computation.

The combined result of these mechanisms is that the GPU sees a large memory space, but when it wanders into a region not currently in VRAM, Trickums swiftly swaps the needed data in. Just as a skilled illusionist manages hidden compartments, Trickums ensures the GPU only notices a slight delay (the latency of a page swap) rather than a hard out-of-memory error. We can visualize page movement akin to sliding puzzle pieces: keep the blank spot moving around to let a new piece in.

Eviction Policy Enhancements: We mentioned LRU as a base policy, but Trickums can integrate smarter heuristics: - **Priority Hints:** Certain data (e.g., active model layers or frequently reused weights) can be given high priority to keep in VRAM. Trickums' API could allow Hellhound or the model loader to tag some allocations as high-priority. These might be evicted last or only if absolutely necessary. - **Proactive Eviction:** Some frameworks proactively release GPU memory when not needed (e.g., after a layer finishes, free its activations). Trickums can hook into such signals to evict those pages immediately and free VRAM for upcoming uses. This works well for static graphs or pipelined models – e.g., once layer 1 is done, its weights could be evicted if needed to load layer 5, etc. - **Access Counters:** As an advanced feature, Trickums could track how many times each page is accessed over a window. If a page is accessed frequently (hot page), a pure LRU might still evict it if there's a period of inactivity, but Trickums might identify it as "don't evict because it will be needed soon again." A hybrid of LRU and LFU (least frequently used) or even machine-learning-based predictors could be used. Researchers have explored using neural networks to predict page access patterns and improve prefetch/eviction decisions beyond simple LRU[17][18]. In a future version, Trickums may employ an AI-driven policy (discussed later under enhancements).

Handling Fragmentation, Alignment, and Bandwidth Considerations

Managing memory in multiple tiers presents practical challenges like **fragmentation** and alignment:

- **VRAM Fragmentation:** Over time, as Trickums allocates and frees varying sizes, VRAM can become fragmented (holes between allocations). Unlike system memory, GPU memory managers cannot as easily move allocations around (especially if pointers are in use by the GPU). Trickums, by virtue of controlling allocations, can mitigate fragmentation by using a paging scheme – since everything is allocated in page-sized chunks, VRAM is essentially managed as a pool of equal-sized blocks. This avoids fragmentation at the cost of internal fragmentation (the last few unused bytes of a page). If a large contiguous VRAM region is needed (e.g., for a single tensor), Trickums can allocate multiple non-contiguous pages and the illusion is maintained by the virtual addressing. Modern GPUs support virtual

addressing, so contiguous virtual memory can map to disjoint physical chunks. Therefore, Trickums can defragment by remapping pages rather than physically moving them (if an allocation can tolerate non-physical-contiguity, which most can thanks to GPU MMU). In rare cases where truly contiguous physical memory is needed (some legacy CUDA operations or certain drivers), Trickums might need to allocate those up front or use a compaction algorithm on evictions.

- **Pinned RAM Fragmentation:** The pinned host memory used by Trickums also needs management. If many allocations are active, Trickums might keep a large pinned region (like a big pool) and sub-allocate from it, again to reduce overhead. Pinned memory is a limited resource (excessive pinning can reduce overall system performance and cannot be swapped by OS), so Trickums might unpin or free host memory for pages that have long since been evicted to disk and are not likely to be needed soon, to relieve pressure on system memory.

- **Alignment:** Alignment is crucial for performance especially on the PCIe and disk transfers. Trickums aligns each allocated chunk to at least 4096 bytes (page size) or more. GPU DMA engines often operate more efficiently on 64-byte or 256-byte boundaries. NVMe best performance is with 4KB alignment and sizes in multiples of 4KB. Trickums thus ensures that the virtual pages align to such boundaries. For instance, if a model weight tensor doesn't naturally align, Trickums may pad it. Also, when using GPUDirect Storage or RDMA, some alignments are required (GPUDirect often requires memory addresses to be pinned and possibly 64KB aligned for peer access in some cases[19]). Trickums takes care of these under the hood.

- **Bandwidth Considerations:** Because Tier-2 and Tier-3 involve relatively slow links (PCIe and SSD), bandwidth can become a bottleneck if we thrash data in and out. Trickums employs a few strategies to handle this:

- **Coalesced transfers:** As noted, page size tuning helps saturate the bandwidth by moving larger chunks. Trickums will try to always use the copy engine on the GPU (DMA) to transfer data asynchronously, allowing overlap with computation. Modern GPUs have multiple copy engines, so Trickums could use one for HtoD (host-to-device) prefetches while the GPU concurrently executes kernels on other data.
- **Compression:** (This will be discussed more in enhancements, but worth noting here too.) Compressing data before writing to disk or sending over network can effectively increase bandwidth at the cost of CPU/GPU cycles for compress/decompress. For example, if large weight matrices have lots of redundancy, compressing them could make transfers faster. Trickums might optionally compress pages evicted to Tier-3 to reduce I/O.
- **Avoiding Redundant Moves:** If a page oscillates between VRAM and RAM repeatedly (ping-ponging), performance plummets. Trickums might detect this pattern and decide to keep it in VRAM (even if it means evicting something else more aggressively) or replicate it on multiple GPUs if that's an issue. Also, if the

GPU is accessing data in a streaming fashion (use once then never again), Trickums can optimize by not caching it in VRAM at all – just have the GPU fetch it via zero-copy from host (which avoids taking up VRAM and subsequent eviction). NVIDIA's Unified Memory can do something like that with "access where it is" strategy on Power9 systems (allowing remote access without migration for infrequent access)[20]. Trickums could similarly decide that some pages stay in host memory if moving them is not worth it (especially if PCIe can handle the throughput).

- **Direct Paths:** As mentioned, Trickums utilizes GPUDirect Storage (if available) to read from NVMe to GPU without copying via CPU. Similarly, for network, using RDMA (GPUDirect RDMA) to fetch data from remote memory straight into local VRAM cuts down on extra copy steps. These direct pipelines maximize the use of available bandwidth.

In essence, Trickums acts as a **virtual memory manager tuned for GPUs**, juggling a hierarchy of memory. Its implementation draws from operating systems (paging, caching, eviction) but is tailored to the characteristics of GPU computing – large contiguous arrays, mostly read-heavy model weights, and the need to overlap communication with computation.

C++ Integration with PyTorch (Shim Layer)

To integrate with PyTorch (and similar frameworks like TensorFlow), Trickums provides a shim that makes the process mostly transparent to the user's code. A typical integration might involve:

- **Custom Allocator:** PyTorch allows custom allocators for CUDA tensors via its `CUDACachingAllocator` interface. Trickums can register itself as the allocator. Thus, when PyTorch does `torch.cuda.tensor(...)` or allocates memory for model parameters, it calls into TrickumsMalloc instead of the default cudaMalloc. This ensures all tensor storage uses Trickums-managed memory. PyTorch's caching allocator normally grabs large chunks of device memory and suballocates to avoid calling cudaMalloc frequently; Trickums might bypass or augment this by doing its own caching at the host side (since it's anyway pooling memory to manage pages). The goal is that PyTorch and the user don't see any difference except perhaps some performance overhead when actual swapping happens.

- **Memory Reports:** One tricky aspect is that PyTorch and CUDA often query available memory (e.g., `cudaMemGetInfo`) to decide how much they can allocate, or to print memory usage. Trickums must intercept these queries too, possibly inflating the reported total memory to the "virtual" size. For example, if a GPU has 8 GB physical, Trickums might report 16 GB available (if it knows it can utilize host memory to that extent). However, this must be done carefully: if we report too large and the program tries to use it fully, performance will degrade significantly due to swapping. Perhaps Trickums reports a moderate multiple of actual VRAM or some number based on configured policy. In any case, lying to `cudaMemGetInfo` is part of the

illusion – making the framework think more VRAM exists. (This addresses frameworks that try to load as much model as fits; now they might load more, relying on Trickums to manage it.)

- **PyTorch Autograd and Streams:** PyTorch uses CUDA streams and events for async execution. Trickums must ensure that page transfers (HtoD or DtoH) are properly synchronized with these. For instance, if PyTorch launches a kernel that will use some data, Trickums should ideally have already prefetched that data into VRAM before the kernel runs. If not, the unified memory page faults will stall the kernel implicitly. Trickums could hook at an even higher level – for example, the model execution graph – with Hellhound's help to schedule prefetches. Alternatively, Trickums might use `cudaStreamAttachMemAsync` or prefetch calls on specific streams to bring in pages at the right time. This requires careful coordination: if PyTorch records an op that will use tensor T on stream S, Trickums could record a prefetch of T's pages into VRAM on stream S, so that the data movement happens before the compute on that stream, possibly overlapping with other streams.

- **C/C++ APIs:** For lower-level usage, Trickums might expose its own API for developers. For example, a function to pin a tensor in VRAM (to exclude it from eviction), or to manually prefetch a tensor. However, in a typical deployment as a daemon/allocator, these might not be needed by end-users – the system "just works" by intercepting standard calls.

- **Daemon & Coordination:** Trickums might run as a background daemon process (especially for handling disk I/O or remote communication) in addition to the interception library in the application process. The two can communicate (via shared memory or sockets) to coordinate page evictions and fetches. For instance, the application thread triggers faults, the Trickums library notifies the daemon "need page X from disk", the daemon does the disk IO and then signals when data is ready, etc. This separation is not strictly necessary but can improve throughput (dedicated threads for I/O and network).

All these implementation measures combine to allow PyTorch models to run **unchanged** on ForgeBorn nodes with Trickums. The metaphoric result: PyTorch thinks it has a "magically" larger GPU. Under the hood, Trickums and Hellhound are furiously shuttling pieces of the model in and out of the GPU like a team of hidden blacksmith's apprentices, ensuring that when the smith (GPU) reaches for a tool (data), it's there, even if moments before it was in the shed.

Comparison to Related Technologies

Trickums' approach to GPU memory virtualization can be better understood by comparing it with existing technologies and techniques aimed at addressing GPU memory limits:

- **NVIDIA Unified Memory (UVM):** Unified Memory allows a programmer to allocate a memory buffer that is automatically managed between CPU and GPU by the CUDA

driver. It implements on-demand page migration and eviction between GPU VRAM and CPU RAM[4]. In concept, this is very similar to Trickums' Tier-1 and Tier-2 management. However, UVM is limited by the size of system memory – each unified allocation is backed by pinned host memory equal to its size, which effectively means it **"cannot go beyond host memory due to page pinning"**[8]. Trickums extends beyond this by adding Tier-3 disk swap, thereby handling oversubscription even when GPU + RAM is not enough. Another difference is control and optimization: UVM's policies are mostly automatic and generic, whereas Trickums can incorporate domain-specific hints (from Hellhound) and custom eviction strategies. For example, UVM might fault pages in purely reactively and evict via LRU, whereas Trickums can proactively prefetch model layers and, say, avoid evicting critical pages (or use knowledge that certain memory is read-only, etc.). In essence, UVM is a general facility (great for ease of programming), but developers have found that automated heuristics can sometimes be suboptimal[21]. Trickums takes a more **active role**: it's like a managed UVM with multi-tier backing and integration with the application's semantics. It's also worth noting Unified Memory incurs overhead on each page fault (GPU context switches to service faults), which can dramatically slow down kernels if many faults occur[22]. Trickums tries to mitigate that with bigger transfers and prefetch. One can think of Trickums as *"UVM on steroids"* – combining hardware page fault support with software-guided policies and extending the concept to disk and network.

- **AMD Smart Access Memory (Resizable BAR):** Smart Access Memory (SAM) is a feature that allows the CPU to directly address the entire VRAM of the GPU (enabled by the PCIe Resizable BAR capability)[23]. Traditionally, CPUs could only map a 256 MB window of GPU memory at a time; SAM removes that limit, letting the CPU read/write all GPU memory. This is essentially a widening of the aperture between CPU and GPU. While not directly about oversubscribing GPU memory, it does relate to unified access. With SAM, the CPU can efficiently stream data to/from VRAM without the overhead of small window management. Trickums can benefit from this: for instance, if SAM is enabled, Trickums writing to the GPU's memory from a CPU thread (for prefetch or eviction) can potentially be faster or simpler (no need for many `cudaMemcpy` calls; it could just write into mapped VRAM addresses). But in the bigger picture, SAM does **not increase GPU memory size**; it just gives the CPU a broader view. If anything, SAM is more analogous to the inverse of Trickums Tier-2: SAM makes VRAM more accessible to CPU, whereas Trickums makes system memory (and beyond) more accessible to GPU. Both unify the address space in different directions. In terms of performance, SAM can improve data transfer rates in some cases (by avoiding some copy overhead)[24][25], but it doesn't implement any paging or eviction. So, Trickums is complementary – one could use SAM in conjunction so that CPU involvement in Trickums memory moves is minimized.

- **Remote CUDA (rCUDA):** rCUDA is a middleware that allows an application to use a GPU over the network as if it were local[9]. It intercepts CUDA calls and forwards

them to a remote server that has a GPU, then results are sent back. The primary use-case is a cluster where not every node has a GPU; a node can "borrow" a GPU from a neighbor. At a high level, Trickums' optional Tier-4 (remote memory) has a similar flavor – it uses network to extend resources. However, rCUDA is about remote **computation** (the kernel actually runs on the remote GPU and data is shipped back and forth), whereas Trickums' remote memory means the local GPU still does the computation, but may fetch some data from a peer. In rCUDA, if a node without a GPU calls `cudaMalloc`, it's actually allocating on a remote GPU and all subsequent `cudaMemcpy`, kernel launches etc., go through the network[10]. In Trickums, when we use remote Tier-4, it might be that `cudaMalloc` was local (Trickums gave a local pointer), but behind the scenes Trickums might store the content on a remote node until needed. One could imagine a hybrid: if the remote node also has a GPU, maybe it could even compute on the data instead of shipping it – but that enters the realm of distributed computing rather than pure memory extension. **Performance:** rCUDA can be surprisingly efficient on fast networks, but it introduces latency on every API call. Trickums aims to keep most execution local to avoid that latency except when swapping large chunks. Another related tech is NVidia's upcoming **GPU Partitioning/Pooling** (which can aggregate multiple GPU memory spaces via NVLink or NVSwitch in a single system). That's more specialized hardware support. Trickums is a software solution that works in more generic settings, albeit with the cost of PCIe/NVMe/Network overhead.

- **NVIDIA GPUDirect (RDMA and Storage):** These are low-level optimizations rather than memory management on their own. **GPUDirect RDMA** allows third-party devices (like NICs) to directly read/write GPU memory. **GPUDirect Storage (GDS)** enables GPUs to perform DMA to storage or through an NVMe controller to GPU memory without bouncing through CPU memory[6]. Trickums doesn't compete with these; it leverages them. For example, with GPUDirect Storage, Trickums can issue a read from the swap file such that the data goes straight into a staging buffer in VRAM. Without GDS, the data path would be: SSD -> CPU RAM -> copy to VRAM, which uses extra bandwidth on the CPU side. With GDS: SSD -> VRAM directly (perhaps via the NVMe DMA engine). This reduces latency and CPU overhead significantly[26]. Similarly, for remote, using GPUDirect RDMA means the NIC can inject data into VRAM directly. If Trickums is running on hardware and drivers that support these features, it can optimize accordingly. Unified Memory in newer CUDA also can utilize some of these under the hood (e.g., UM on NVLink or NVSwitch can directly access other GPU memory). The main point: GPUDirect is about efficient data movement, whereas Trickums is about deciding **what** data to move **when**. They complement each other in achieving the overall goal.

- **AMD ROCm Unified Memory / HMM:** AMD's ROCm has a unified memory concept using Linux Heterogeneous Memory Management (HMM) which is similar to NVIDIA's. It allows page migration and CPU-GPU coherent memory on AMD GPUs (especially MI200 Instinct GPUs)[27][28]. The differences mirror NVIDIA's case:

AMD's unified memory won't inherently use disk or remote, it's within a single node's CPU+GPU memory. AMD's SmartAccess (discussed above) and features like large BAR make certain operations faster but do not provide oversubscription beyond host memory either. Trickums could in theory work on AMD GPUs as well, using their APIs to pin memory and handle page faults (if accessible via HMM events). The concept stays the same.

In summary, **what sets Trickums apart** is its **holistic approach**: it combines ideas of unified memory, caching, and distributed memory into one system explicitly designed for AI workloads on a decentralized network. Where each related technology addresses a piece (automatic memory oversubscription, or remote GPU use, or faster I/O), Trickums integrates them with a guiding narrative – it is aware of the AI model structure (through Hellhound) and network topology, and it prioritizes usage patterns accordingly. The metaphoric difference: others are tools or features (like a single enchantment), while Trickums is an entire spellbook orchestrating an illusion – from small sleights of hand (like prefetching pages) to grand tricks (swapping across the network).

Integration with ForgeBorn Daemons (Hellhound & Trickums Collaboration)

Within the ForgeBorn architecture, Trickums does not operate in isolation; it works in concert with other daemons to maximize efficiency. **Hellhound** is another core daemon in ForgeBorn (described in the Genesis scroll) – conceptually, Hellhound could be the orchestration demon that manages task scheduling, model partitioning ("sharding"), and workload prediction. The cooperation between Hellhound and Trickums can be imagined as between a strategist and a quartermaster: Hellhound decides *what* parts of the model will be needed and when, while Trickums figures out *where* to fetch them from and *how* to store them.

Here are specific ways Hellhound and Trickums integrate:

- **Model Shard Management:** ForgeBorn likely breaks large AI models into "shards" (e.g., different layers or blocks of a neural network, or different attention heads, etc.) which can be distributed across nodes or loaded/unloaded dynamically. Hellhound might be responsible for this partitioning – for example, deciding that Node A will handle layers 1-5 of a model and Node B will handle 6-10, or even within one node, deciding which chunks of the model to keep in memory at a time. Trickums provides the memory substrate to back this. Hellhound, knowing the model's execution order, can inform Trickums ahead of time: "After the current layer, the next shard needed will be X (currently on disk or on a peer); prepare it." This is essentially **predictive prefetching** at the high level. Instead of waiting for a page fault, Trickums can proactively pull in entire shards (several pages worth) when cued. Hellhound could communicate with Trickums via an API or shared memory signals, enumerating which model parts (e.g., weight matrices, activations) will be required in the next timestep or for the next request.

- **Prefetch Daemon:** Hellhound might run as a background thread (or separate process) monitoring the incoming inference requests or training steps. For instance, consider an LLM (large language model) being served: Hellhound sees the sequence of layers that will run for each token. As token generation proceeds, it can always ensure the next layer's weights are prefetched into VRAM (or at least into host memory ready to DMA) by the time the current layer finishes. Trickums exports functions like `prefetch_to_gpu(void* ptr, size_t bytes)` which Hellhound can invoke to stage data. This way, ideally, when the compute engine (the GPU) is ready to consume layer N+1, the data is already in Tier-1 or Tier-2, avoiding a stall. This tight coupling could drastically reduce the overhead of virtual memory. It's essentially overlapping model loading with computation – a form of pipeline.

- **Hinting Evictions:** Hellhound's global view also helps decide what can be evicted. For example, if an entire model's forward pass is done, the activations from layer 1 won't be needed until possibly backprop (in training) or never again (in inference). Hellhound can mark those as evictable. Trickums can then eagerly evict or at least mark them low priority. Another scenario: if Hellhound decides to reroute a certain model shard to a different node (for load balancing), it can inform Trickums that "shard Y can be dropped from local memory after this batch, we won't use it again soon." Trickums can then free up VRAM for other things sooner.

- **Remote Coordination (VAIPU net):** When using Tier-4 remote memory, Hellhound likely plays a role in negotiating that. Perhaps Hellhound on Node A communicates with Hellhound on Node B (or a central coordinator) to request memory space. For instance, Node A might say, "I'm low on local RAM, can someone hold shard Z for me?" Node B might volunteer and Hellhound sets up a channel. Trickums then will know that shard Z's home is Node B and will pull it via RDMA when needed. Hellhound could also decide to instead move computation to Node B (if Node A is too constrained), which is an alternate approach (moving compute to data rather than data to compute). The design likely emphasizes decentralization: so perhaps the preference is to move data since each node might be doing part of the work. The interplay is dynamic and could even be market-based (if ForgeBorn has an economic aspect, maybe nodes "rent" memory to each other).

- **Synchronization and Consistency:** With Hellhound guiding high-level decisions, Trickums must ensure consistency of data. If multiple nodes might cache the same shard, and if that shard updates (e.g., training scenario where weights change), Hellhound would be responsible for invalidating old copies or updating them. Trickums could include validation checks (like version numbers on pages) so that stale data isn't accidentally used. For inference (read-only weights), consistency is simpler – replication is fine as long as everyone has the same copy.

In narrative terms, Hellhound could be depicted as a fiery hound that runs ahead on the path, scouting what lies before the forge's warriors, while Trickums is the trickster forging illusions so the army (the GPUs) always find the weapons (data) they need at hand. The

cooperation ensures that even if an individual node's memory is limited, the network as a whole can bear the load: Hellhound coordinates, Trickums delivers. This forms the basis of ForgeBorn's **decentralized AI compute** – by dividing and conquering memory demands among many collaborators, they achieve what a single GPU couldn't.

A concrete example: Suppose ForgeBorn is serving a massive 70B-parameter language model across several 8GB GPUs. Hellhound might assign each GPU a certain subset of layers primarily. When a request comes in, each GPU will handle its layers and then pass intermediate results. But even within each GPU, 8GB might not hold all its assigned layers at once. So Hellhound instructs Trickums to load layers 1 and 2 first. GPU runs them, then while GPU is working on layer 2, Hellhound tells Trickums to start loading layer 3 (which was on disk or another node). By the time GPU needs layer 3, Trickums has it in VRAM (or at least in RAM ready to go). Meanwhile, layer 1's weights can be evicted to free space. And so on. The Hellhound-Trickums duo essentially perform layer-by-layer streaming, making sure the GPU is busy computing rather than idly waiting on data. This kind of synergy between scheduling (Hellhound) and memory virtualization (Trickums) is what allows ForgeBorn to **"operate across low-memory GPUs"** effectively.

Pseudocode and System Design of the Trickums Runtime

To illustrate the design at a lower level, here we present a more cohesive pseudocode outline of Trickums' runtime components. This is a conceptual design merging the ideas discussed:

```
// Data structures (simplified)
struct PageInfo {
    void* hostPtr;
    bool  inVram;
    size_t vramOffset;
    bool  dirty;
    Timestamp lastAccess;
};
struct AllocationMeta {
    void* basePtr;
    size_t totalBytes;
    size_t pageSize;
    std::vector<PageInfo> pages;
};

// Global state
VramManager vram;    // manages VRAM blocks (bitmap of free 4MB chunks, etc.)
DiskSwapManager disk; // manages swap file offsets
LRUList<PageInfo*> lruList; // global LRU for pages in VRAM
std::map<void*, AllocationMeta*> allocMap; // maps basePtr to meta

// Fault Handler (invoked when GPU page fault occurs or on manual prefetch
request)
void handlePageFault(void* pagePtr) {
```

```
    AllocationMeta* meta = findAllocForPtr(pagePtr);
    size_t pageIndex = ( (char*)pagePtr - (char*)meta->basePtr ) / meta-
>pageSize;
    PageInfo &page = meta->pages[pageIndex];
    std::lock_guard<std::mutex> lock(meta->mutex);
    if (page.inVram) {
        // Already in VRAM (maybe another thread handled it first)
        return;
    }
    // Need to bring page into VRAM
    // Step 1: Find free VRAM slot, evict if necessary
    size_t freeSlot = vram.findFreeChunk();
    if (freeSlot == NO_FREE) {
        PageInfo* evictPage = lruList.ejectLRU();
        evictPageFromVram(evictPage);
        freeSlot = evictPage->vramOffset;
    }
    // Step 2: Load data into that slot
    if (pageIsOnDisk(page)) {
        disk.read(page, tempHostBuffer);  // read page from disk to
tempHostBuffer
        cudaMemcpyAsync(VRAM_ADDR(freeSlot), tempHostBuffer, meta->pageSize,
HtoD);
    } else {
        // page is in host RAM (pinned), can DMA directly
        cudaMemcpyAsync(VRAM_ADDR(freeSlot), page.hostPtr, meta->pageSize,
HtoD);
    }
    cudaStreamSynchronize(0); // wait for copy to complete (or use events)
    // Step 3: Update metadata
    page.inVram = true;
    page.vramOffset = freeSlot;
    page.dirty = false; // assume loaded data is up-to-date (weights or zero-
inited)
    page.lastAccess = now();
    lruList.insert(&page);
}

// Evict a page from VRAM to host or disk
void evictPageFromVram(PageInfo* page) {
    // Called when needing to free a VRAM slot
    if (page->dirty) {
        // Write back to host or disk
        if (hostMemoryAvailable()) {
            // move to host pinned memory
            cudaMemcpyAsync(page->hostPtr, VRAM_ADDR(page->vramOffset),
PAGE_SIZE, DtoH);
            // (if host is full, we might directly write to disk from VRAM if
possible)
        } else {
```

```
            // host RAM full, write to disk
            void* swapBuf = getTempBuffer();
            cudaMemcpy(swapBuf, VRAM_ADDR(page->vramOffset), PAGE_SIZE,
DtoH);
            disk.write(page, swapBuf);
        }
    }
    page->inVram = false;
    // Free VRAM chunk
    vram.freeChunk(page->vramOffset);
    // Remove from LRU list, etc.
}
```

This pseudocode is highly simplified (synchronous, no error handling, single stream assumption), but it captures the essence: - `handlePageFault` finds which page is needed, ensures a VRAM slot by possibly evicting something else (using LRU), then loads the page from wherever it is (RAM or disk) into VRAM, and updates metadata. In practice this would be triggered by an actual page fault signal from CUDA or by a prefetch call. - `evictPageFromVram` handles writing a page back out if needed and marking it not in VRAM. Note that if the page wasn't modified (`dirty==false`), and if it's just weights from disk, we could skip writing back – we could just drop it knowing we have a copy on disk. The code above is conservative (always writes back if dirty flag set).

The Trickums runtime would have multiple threads: one could be an asynchronous pager (servicing page faults or prefetch requests), one could handle disk I/O completion, etc., to avoid stalling the main thread. Modern OS integration (like using the Linux `uffd` userfaultfd mechanism or Windows' equivalent) could allow user-space handling of page faults; NVIDIA's driver doesn't expose page faults directly to user, but unified memory does behind the scenes. If Trickums is built in user space, it might rely on UVM to trigger data movement and just guide it. If built as a kernel module or driver layer, it could intercept at a lower level.

Security and Correctness Checks: The pseudocode above omits them, but Trickums must ensure: - Memory bounds are respected (no reading beyond allocated regions – likely handled by hardware page protections). - GPU kernels do not read uninitialized data – Trickums might zero-fill pages on first allocation if needed (just like an OS zeroes pages for security). - Data in transit to remote nodes might need encryption for security if going over untrusted networks (ForgeBorn being decentralized implies possibly untrusted peers). Trickums could integrate encryption/decryption for any sensitive data leaving the local node, to prevent snooping. This adds overhead but can be offloaded (maybe via GPU encryption engines or using NIC offload). - If a remote node fails or disconnects while holding some memory pages, Trickums/Hellhound need contingency: perhaps replication of critical data on more than one node, or quickly recompute/replace missing data from disk if possible. For example, if a remote memory tier was being used as cache for disk data, losing it means you can fallback to local disk copy. - Validation of data integrity: using checksums for pages on disk or received from network ensures correctness. Trickums

might store a hash for each page on disk such that when reading back, it can detect corruption (rare but possible in case of disk error). Similarly, network packets could be corrupted; CRC at lower layers helps, but end-to-end checksum could too. - Deadlock avoidance: If Trickums requires GPU to copy something but GPU is busy waiting on Trickums (like if all copy engines are tied up), careful ordering and using separate streams is needed. The design ensures that a page fault handling uses a separate CUDA stream for HtoD copy that can execute even if default stream has a running kernel (because otherwise you'd have a catch-22: kernel waiting for data, but data copy queued behind that kernel on same stream). So Trickums uses either the dedicated copy stream or leverages the UVM's ability to page in concurrently.

All these details make Trickums a fairly complex system, almost an OS within an OS for GPU memory. A rigorous testing and validation phase is needed – injecting known access patterns and ensuring data output matches a baseline (to catch any scenario where Trickums might, say, accidentally drop a needed page or not load something in time causing incorrect results or crashes).

Security, Correctness, and Validation

Operating at such a low level with memory, Trickums must maintain strict correctness and consider security in a decentralized environment:

- **Memory Safety:** Trickums never exposes memory belonging to one process/node to another unauthorized process. In a multi-tenant scenario, each client's allocations are tracked separately. Even though Trickums intercepts calls globally, it tags allocations by which client or model they belong to, and isolates their address spaces. This is akin to an OS having separate page tables per process. In ForgeBorn, if multiple models or users share a GPU, Trickums ensures one user's data swapped to disk is encrypted or at least not accessible by another user. On a single user scenario, it ensures pointers are valid – any access outside allocated ranges should ideally trigger a fault and be handled gracefully (or be caught and reported).

- **Data Encryption:** As mentioned, Tier-4 remote memory usage raises trust issues: you might not fully trust a peer to hold your model weights (they might copy them). For economic inclusivity, maybe models are community property, but if not, Trickums could integrate encryption for pages before sending to a remote cache. The remote node then just stores cipher text and returns it; the local node would decrypt upon retrieval. This could use symmetric encryption with keys managed by the owner. There's a performance cost, but for highly sensitive data it may be worth it. Alternatively, secret-sharing the model shards or splitting across peers so no single peer has a usable piece of the model could be an approach (beyond Trickums scope, perhaps Hellhound level trust management).

- **Validation & Checksums:** To ensure no data corruption, Trickums can maintain a checksum for each page's content (particularly for disk pages). When writing to disk, store a hash; on reading back, recompute and verify. This catches disk errors or stray writes. In network transfers, built-in CRC and our own end-to-end hash can ensure data integrity across the wire. It's important because a single bit flip in a model weight could cause weird issues in inference; better to detect and maybe re-send the data.

- **Resource Limits and DoS:** Trickums should enforce quotas so that a single user doesn't consume infinite disk space or memory beyond intended. It likely has configurable limits for how much disk swap to use, how much remote memory to borrow, etc. If multiple processes use Trickums, it should prevent thrashing the system into the ground (for example, two heavy oversubscribers could continuously swap and saturate the disk, making everything slow). Hellhound, as a scheduler, can also mitigate by not over-committing memory beyond what Trickums can handle with some baseline performance.

- **Correctness in Eviction Logic:** The algorithm must avoid evicting something that is currently in use. Trickums uses reference counting or pin counting for pages: e.g., if a kernel is actively using page X (we can know if a page was faulted in and the kernel hasn't finished), we mark it as in-use so it's not evicted mid-use. Typically, page fault handling in CUDA ensures this by design: it won't evict a page until a kernel is done with it. Trickums should coordinate with such mechanisms or add its own locks. Also, multi-GPU cases (if one GPU could access memory of another via NVLink or CPU pointers) complicate it, but likely ForgeBorn uses one Trickums instance per GPU device.

- **Testing Approach:** A robust test would involve running known workloads with Trickums and verifying results match non-Trickums runs. Memory torture tests (allocating more than VRAM and writing patterns to memory to ensure they read back correctly) will be part of validation. Also performance benchmarking to ensure that the overhead is manageable (Trickums might allow e.g. 2x model size with 50% slowdown; if it were 10x slowdown that might not be acceptable for some use-cases – tuning needed).

In the narrative: Trickums' illusions are potent, but must be handled with care lest the illusion falter. Thus, there are safeguards (magical wards, if you will) to ensure the deception never leads to actual chaos – data lost in the ether or secrets stolen by eavesdroppers.

Use Cases and Deployment Scenarios

The Trickums system unlocks several practical scenarios in the ForgeBorn ecosystem:

- **Running Large Models on Consumer GPUs:** This is the flagship use-case. For instance, imagine a researcher with a gaming GPU (say 6 GB VRAM) wanting to run a

transformer model that normally requires 16 GB. With Trickums, they can load the model (perhaps 12 GB weights plus overhead) and Trickums will automatically swap parts in and out. The researcher simply runs their PyTorch code; behind the scenes, Trickums might be moving layers to host RAM and back. The cost is some slowdown due to transfers, but if the model is primarily for inference and can tolerate some latency, this enables access to models that were previously out of reach. It fosters **economic inclusivity**: you don't need a $3000 GPU to participate in AI tasks – even a cheaper card, supplemented by your system RAM and fast SSD, can contribute. For example, Stable Diffusion image generation on a 4 GB card could generate high-res images by swapping UNet weights layer by layer, albeit a bit slower. Community demos have shown even without Trickums, it's possible by manually offloading layers; Trickums automates and generalizes that process.

- **Inference-Only VAIPU Nodes:** In a decentralized network, not all nodes will do training or heavy multi-GPU tasks. Some might be inference specialists – e.g., an edge server or a user's desktop that only performs forward passes on models to serve answers. These **VAIPU (Virtual AI Processing Unit) nodes** might have minimal GPU or none at all. With Trickums, even a node with just a CPU could host a large model in a hybrid CPU-GPU way: perhaps the CPU holds most of the model in RAM and only small chunks are sent to a tiny GPU for acceleration on critical parts. Or if truly no GPU, Trickums might not apply (it's GPU memory virtualization), but those nodes could serve as remote memory providers to others. More interestingly, an inference node with a small GPU can leverage remote memory from a neighbor: for instance, a node with a 4 GB GPU can borrow another 4 GB over the network from a neighbor's idle VRAM, effectively acting like an 8 GB GPU logically. In practice, this could mean the difference between being able to run a particular model or not. Latency is critical in inference though: one has to design the model execution to hide the latency of fetching remote chunks (which Hellhound's prefetch can help with). An inference-only node could also dedicate more of its resources to caching model data rather than doing backprop or such, which fits Trickums well.

- **Distributed Training with Swap:** Training large models usually involves multi-GPU setups or model parallelism. Trickums could facilitate a form of model parallelism where each GPU gets a part of the model and swaps layers in/out when needed. However, training has the complication of backward pass requiring gradients and possibly needing the weights again. Trickums could still help by offloading optimizer states or gradients to CPU memory between iterations, etc. There is a known approach called ZeRO-Offload (in Microsoft DeepSpeed) that offloads optimizer memory to CPU to allow training bigger models on limited GPU memory. Trickums could be used to implement something like ZeRO-Offload in a general way: all those additional tensors (optimizer, momenta, etc.) can be kept in host memory and paged in when required for update step, then paged out. This would allow, for instance, training a model that needs 20 GB of memory on a GPU with 10 GB, by

leveraging 16 GB of system RAM for the optimizer states and activations, at some performance cost.

- **Remote-Swap Setups and Memory Sharing:** In a scenario where a few machines are connected (a small cluster or even ad-hoc between friends across the internet), remote-swap can shine. For example, suppose one machine has a powerful GPU with low VRAM (say a Tesla T4 with 16 GB) and another machine has a mid GPU with 8 GB but lots of free RAM or a spare GPU that's idle. The first machine could use the second as a "swap server" – instead of hitting its disk, it goes over 10 Gbit Ethernet to the second machine's memory. If that network is fast enough (and latency ~ say 1ms), this might beat an NVMe drive on latency (which might be 0.1ms, actually NVMe is quite low latency and 1ms would be slower; but multiple outstanding requests could hide latency). In any case, in a cluster with a fast fabric (InfiniBand, NVLink P2P if within a node, or PCIe peering), sharing memory can yield interesting possibilities:

- A form of **GPU memory pooling**: some recent HPC systems allow pooling GPU memory via NVSwitch such that one GPU can use memory attached to another GPU almost as if it was local (with NVSwitch providing high bandwidth). Trickums could mimic that in software over network or PCIe: not as fast, but conceptually similar.
- **High availability**: if one node's disk is slow, but another node's disk is a super-fast NVMe, perhaps even that could be leveraged (though at that point, just put NVMe in each node for local).

A remote-swap setup might also be useful in cloud or container environments: one container can use memory from another physical host in the cluster – effectively disaggregating memory from compute. This aligns with trends in data centers to disaggregate resources (so you could independently scale GPU compute and memory capacity). Trickums could be part of that software stack enabling memory disaggregation for GPUs.

- **Edge Computing and IoT:** In edge scenarios, you might have GPU-equipped devices that are memory-constrained (like a Jetson Nano with 4GB). Trickums could allow such a device to still run larger models by streaming data from a central server. For instance, an AR headset with a small GPU could run an AI model by fetching weights on-the-fly from a nearby edge server when needed. The trade-off is latency and connectivity, but if done intelligently (prefetching based on sensor context – e.g., the AR app guesses which AI tasks or model parts will be needed next), it can work. This is an advanced use-case but shows the possibility of Trickums enabling **compute anywhere** – you are not barred from running something just because of memory, as long as you have some form of storage or network.

In all these scenarios, performance will vary. Trickums is not a magic that makes an 8GB card as fast as a 16GB card with a heavy model – there will be slowdowns due to data

transfer. However, the key is that it makes it **possible** at all, and often the slowdowns can be mitigated by overlap and smart scheduling. For instance, if an inference pipeline is well-optimized, maybe running a 2× model on half memory could still achieve 50-70% of the throughput of the full-memory scenario (just hypothetical). The user must balance memory vs speed, but Trickums gives that flexibility.

Deployment wise, Trickums would be installed as part of the ForgeBorn client on each node. Likely it requires a kernel driver or at least admin privileges to lock memory and tune system for minimal paging interference (one wouldn't want the OS to swap pinned memory out to disk – pinned usually means it won't, but it consumes RAM so OS should have enough headroom). It could also come with a monitoring tool to observe how much it's swapping (so advanced users can realize if they're thrashing too much, maybe they choose a smaller model or upgrade hardware).

Future Enhancements (Compression, Sparsity, AI-driven Policies)

Trickums as described is already powerful, but there are exciting avenues to enhance it further:

- **Memory Compression:** Compressing data can effectively increase the capacity of each tier and reduce transfer times, at the cost of compute overhead for compress/decompress. There are a few angles:
- Lossless compression (e.g., LZ4 or ZSTD) on pages before writing to disk or sending over network. If model weights contain redundant patterns or zeros, this could reduce size. Even a 2:1 compression means half the disk IO and network traffic, which is significant. The overhead of compressing a 4MB page might be a few microseconds (if using fast algorithms and possibly hardware accelerators). NVIDIA's Nsight might identify patterns to exploit as well.
- Lossy compression or quantization specifically for model weights: e.g., if using 16-bit floats, maybe compress to 8-bit on disk and then convert back to float16 on load. Or more exotic: store differences between weights (which might be smaller entropy). However, lossy could affect model accuracy, so likely not unless done carefully (some frameworks do allow running models in lower precision to save memory though).
- **Compressed Page Cache:** Another idea is, similar to some OSes (like Windows compresses memory in RAM before swapping to disk), Trickums could compress pages when moving from VRAM to RAM if the overhead is acceptable. GPUs themselves have compression tech (like delta color compression in graphics) but for general memory not so much. Perhaps some pages like activation tensors could compress well (sparsity or low entropy after ReLUs). This is speculative but worth researching.

- **Exploiting Sparsity:** Many deep learning models have sparse patterns: e.g., large embedding matrices where not all vectors are used for a given batch, or activation maps with zeros. Trickums could integrate a **sparse paging** mechanism: rather than

moving full dense pages, it only moves the non-zero elements or chunks actually needed. For example, if a model has a giant embedding table and an inference only touches 5% of it, Trickums doesn't need to swap in the whole thing, just the parts being accessed. This requires understanding of data structures (maybe via hooks in the framework – e.g., know which indices will be looked up and only fetch those). It's more of a model-specific optimization, but one that could drastically cut memory usage for some cases. Another aspect is storing sparse pages efficiently on disk (only store non-zero and an index map). If a model is pruned and 30% of weights are zero, maybe that compresses anyway, or we could avoid moving those zeros at all by marking them and skipping.

- **AI-Driven Prefetch/Eviction:** Instead of fixed heuristics like LRU, an AI model could be trained (offline or online) to predict which pages will be needed and which can be evicted. The research we referenced earlier proposed using RNN or Transformer models on memory access traces[14][29]. Trickums could incorporate a lightweight predictor that classifies pages or sequences of accesses. For instance, observing the last few layer accesses or last few batches' pattern, it might predict "after using page 5, 7, 3, we likely need 6 next" – so prefetch page 6. Or in eviction, it might predict a page that hasn't been used in a while but will be used again soon (so don't evict it, evict another that truly won't be used). Over time, such a system could adapt to different models: e.g., CNNs have sequential layer access (easy to predict), whereas something like a Transformer might revisit certain weights (maybe in attention blocks sharing keys/values) – a learned predictor might catch those nuances. The cost is complexity and overhead of running the predictor; but maybe a small neural net on the CPU could run asynchronously. If ForgeBorn nodes have some idle CPU cores, dedicating some to smarter memory management could pay off in higher effective throughput.

- **Hardware Integration:** In the future, Trickums concepts might be integrated into GPU hardware/drivers. For now, it's a software overlay. But one could imagine a specialized NVMe SSD with GPU-side compression, or NIC that knows about Trickums pages and caches them, etc. If ForgeBorn grows, maybe even a custom "ForgeBorn co-processor" could assist (this goes beyond our scope, but fun to imagine).

- **Better Remote Coordination:** Perhaps dynamic load balancing where if one node is constantly pulling from another's memory, maybe they should just transfer some of the compute or permanently migrate that part of model to the first node's disk and stop fetching repeatedly from remote. Hellhound likely can handle that: after a period, it might say "it's cheaper to just copy shard Z over to my disk so I use that instead of bothering node B each time." Trickums would then repoint to local storage. This blur between memory and placement is interesting – memory virtualization could lead to discovering that some models should be re-sharded for efficiency.

- **User Controls and Telemetry:** Exposing more controls: a user might set a policy like "use at most X GB of my disk for swap" or "prefer to use remote memory if available to save my SSD's lifespan" (SSDs wear out with too many writes – something Trickums should consider by maybe using mostly read from disk and minimal write, or using RAM as write-back cache to reduce SSD writes). Telemetry could show how much data is being moved, average latency of page faults, etc., to give insight and allow tuning. In a decentralized economy, this might even feed into pricing: e.g., if you borrow memory from peers a lot, you pay tokens; if you use your disk instead, no cost but you accept slower speed.

Finally, framing it back in the narrative: these enhancements are like finding new alloys and enchantments to further strengthen the ForgeBorn arsenal – compressing memory like folding steel, skipping zeros like an arrow finding the gaps, and even employing predictive magic to foresee needs. With Trickums continuously evolving, the "VRAM illusion" can only become more convincing, inching closer to making the hardware limits disappear entirely from the user's perspective.

Conclusion

Trickums, the VRAM illusion layer of ForgeBorn, stands as a compelling marriage of metaphor and engineering. It extends the notion of memory beyond physical constraints: through a hierarchy of **forges and phantoms** – from the red-hot GPU VRAM to the cool expanse of system RAM, down to the deep vaults of disk, and across the ether to allied nodes' memory. The system's design draws on proven concepts (paging, caching, unified memory) and innovates by orchestrating them in a distributed, AI-informed context. By intercepting and managing memory at every turn, Trickums deceives the harsh reality of limited VRAM, presenting instead a boundless vista of memory where large models roam freely.

In practice, Trickums enables a form of democratized AI compute: one where a hobbyist's PC with a mid-range GPU can contribute to or benefit from the same AI models that traditionally demanded enterprise hardware. It levels the field by leveraging what's abundant (disk, network, system RAM) to make up for what's scarce (GPU RAM). The cost – some added latency or complexity – is tempered by careful planning (Hellhound's foresight, asynchronous transfers) and thus kept in check.

As we look ahead, Trickums could very well be the cornerstone that allows the ForgeBorn network to scale elastically. It provides the backbone for **economically inclusive AI**, where nodes of varying capability can join forces, share resources, and ensure that knowledge (models) flows to wherever there's compute available, without hitting a memory wall. This companion whitepaper to the ForgeBorn Genesis Scroll has delved into the technical depths of Trickums, but through the lens of narrative – we've seen Trickums as the illusionist, the blacksmith's apprentice, the sorcerer of memory. In doing so, we hope the design is not only clear in engineering terms, but also vivid in concept.

Ultimately, Trickums is about **empowerment**: empowering hardware to do more than its specs, empowering individuals to participate in AI at scale, and empowering the ForgeBorn collective to wield "the fire of the forge" – the massive power of AI models – without being extinguished by practical limits. Through tempered design and clever deception, Trickums keeps the forge flames alight, ensuring that no GPU, however small, is left behind in the quest to forge intelligence from data.

Sources: The design of Trickums builds upon prior art and concepts such as NVIDIA's Unified Memory and GPUDirect technologies[4][6], remote GPU virtualization frameworks like rCUDA[9], and research into GPU memory oversubscription[8][13]. Pinned host memory provides a CPU-GPU bridge albeit with lower bandwidth[2], and Resizable BAR (Smart Access Memory) shows the benefits of broadening CPU-GPU addressability[23]. By integrating these ideas and extending them with a tiered approach and distributed awareness, Trickums creates a unique solution to push beyond the traditional VRAM limits. The cooperation with ForgeBorn's Hellhound for prefetching model shards is a novel layer, aligning with suggestions that smarter (even ML-driven) policies can significantly reduce thrashing and improve utilization[17][18]. In sum, Trickums is an embodiment of multiple state-of-the-art strategies unified under one "roof" to serve the ForgeBorn vision.

---

[1] Resource Management Concepts | Kinetica Docs

https://docs.kinetica.com/7.1/rm/concepts/

[2] [3] [27] [28] GPU memory — ROCm Documentation

https://rocm.docs.amd.com/en/docs-6.0.0/conceptual/gpu-memory.html

[4] [12] [19] [20] [22] Improving GPU Memory Oversubscription Performance | NVIDIA Technical Blog

https://developer.nvidia.com/blog/improving-gpu-memory-oversubscription-performance/

[5] [6] [26] GPUDirect Storage support for IBM Storage Scale

https://www.ibm.com/docs/en/storage-scale/5.2.2?topic=architecture-gpudirect-storage-support-storage-scale

[7] [8] PowerPoint Presentation

https://sc18.supercomputing.org/proceedings/tech_paper/tech_paper_files/pap194s5.pdf

[9] [10] rCUDA - Wikipedia

https://en.wikipedia.org/wiki/RCUDA

[11] Role of GPUDirect RDMA & RoCE in Optimized Paths

https://wolfadvancedtechnology.com/role-of-gpudirect-rdma-roce-in-optimized-paths/

[13] Oversubscribing GPU Unified Virtual Memory: Implications and Suggestions

https://research.spec.org/icpe_proceedings/2022/proceedings/p67.pdf

[14] [15] [16] [17] [18] [29] arxiv.org

https://arxiv.org/pdf/2204.02974

[21] Does CUDA unified memory support LRU or LFU eviction policies when moving data between host and gpu? - CUDA Programming and Performance - NVIDIA Developer Forums

https://forums.developer.nvidia.com/t/does-cuda-unified-memory-support-lru-or-lfu-eviction-policies-when-moving-data-between-host-and-gpu/324154

[23] [24] [25] How to get the most out of Smart Access Memory (SAM) - AMD GPUOpen

https://gpuopen.com/learn/get-the-most-out-of-smart-access-memory/