ChatGPT

Exploring High-Performance GPU Kernels and Other Top Codebases

AMD's $100K GPU Kernel Challenge (Instinct MI300)

Earlier in 2025, AMD launched a **Developer Challenge** with **$100,000 in prizes** to spur development of ultra-fast GPU kernels on the new **Instinct MI300** accelerator [1]. Contestants worldwide optimized key **LLM inference kernels** – such as FP8 matrix multiplication (GEMM), multi-head attention, and fused MoE (Mixture-of-Experts) – targeting MI300 GPUs [1]. Submissions were run on AMD hardware (often using OpenAI's Triton language for GPU kernels) and ranked by speed. The response was huge: **over 30,000 kernel submissions from 163+ teams** poured in over the 2-month contest [2]. Impressively, the best kernels **outperformed AMD's own baseline implementations** on these tasks [3], showcasing how far low-level optimization can go. AMD even highlighted this success at their Advancing AI event, thanking the GPU community for pushing MI300's performance frontier [4] [2].

Open-Source GPU Kernel Repos from the Challenge

Several top competitors from the AMD challenge have open-sourced their **winning kernel code**, providing great learning material for GPU programming enthusiasts:

- **ColorsWind's "RadeonFlow Kernels" (Grand Prize)** – An AMD MI300 kernel suite (HIP C++ and PyTorch) that won the grand prize by achieving *8×+ speedups* over AMD's PyTorch baselines [5]. It includes highly tuned implementations of FP8 **GEMM**, a fused MoE kernel, and an MLA (latent attention) kernel for MI300X GPUs [6]. (See the project's technical report for details on blockwise tiling, memory hierarchy optimizations, etc.) [7] This codebase is a goldmine to study how expert developers manage thread blocks, shared memory, and vectorized operations to maximize throughput on AMD hardware.

- **Luong The Cong's FP8 Quantized MatMul** – An open-source solution focusing on *block-wise FP8 quantization* for matrix multiplication [8]. This was one of the top submissions in the AMD/GPU-MODE challenge. The repository demonstrates how to implement **FP8 GEMM with block scaling** and even provides extras like code to quantize PyTorch BF16 matmuls down to FP8 [8]. It's a great example of precision trade-offs and custom kernel code on GPUs.

- **Snektron's FP8 Matrix-Multiplication Kernels** – Another strong contender's submission (available as a GitHub repo) containing their approach to FP8 GEMM on AMD GPUs [9]. Snektron is known in the OpenCL/HIP community, and their code (likely using Triton or inline HIP) is useful to compare different optimization strategies. While the repo's description is brief ("My submission for the GPUMODE/AMD FP8 MM challenge" [10]), the implementation itself can be studied for techniques like memory tiling, vector loads, etc., on MI300.

- **Seb V's Fast GPU Matrix Blog** – Rather than a repo, this is an **educational blog post** by Sebastien V. (one of the top participants) walking through step-by-step optimizations of GPU matrix multiply on AMD RDNA3 GPUs [11]. He demonstrates how he beat AMD's rocBLAS library by 60% on a 4096×4096 SGEMM, iteratively refining the kernel across 8 versions [11]. Reading

this will give you insight into **general GPU optimization concepts** (memory coalescing, loop unrolling, using LDS/shared memory, wavefront scheduling, etc.) in a narrative form – highly recommended alongside copying code.

- **Akash Karnatak's Challenge Solutions** – Another top finisher who wrote a detailed technical blog about his FP8 GEMM optimizations on MI300 (covering tiling, LDS bank conflict avoidance via swizzling, "stream-K" persistent thread block scheduling, etc.). His write-up (and linked code repo) shows how he approached the kernel in stages, optimizing global-memory to LDS loads, MFMA usage, and write-back [12] [13]. This resource complements the others by revealing the "thought process" behind the code.

(These AMD challenge repos are advanced; don't worry if the code is dense. You might start by reading their READMEs or technical reports first to understand the high-level strategies, then proceed to type out the code to see those strategies in action.)

Renowned CUDA/Open-Source GPU Libraries to Learn From

Beyond the AMD contest, you can learn a ton by exploring some **famous CUDA and GPU computing codebases**. These are high-quality, widely-used projects that illustrate best practices in GPU kernel development:

- **NVIDIA CUTLASS** – A flagship open-source library of **CUDA C++ template components for high-performance linear algebra**. CUTLASS provides a collection of optimized building blocks for GEMMs (matrix multiplies) and other tensor operations, using hierarchical tiling and warp-level primitives [14]. It's essentially how NVIDIA engineers implement things like cuBLAS under the hood. Studying CUTLASS will expose you to warp-level matrix math, **tensor core usage**, and template meta-programming for kernels. (Even if the C++ is complex, reading the simpler examples and docs can be very educational.) [14]

- **NVIDIA CUDA Samples** – NVIDIA's official repository of **CUDA example programs** showcasing various GPU programming features [15]. This is a great starting point for practice: it includes dozens of self-contained samples (vector add, convolution, matrix multiply, graphics interop, etc.) with well-commented code. Copying and running these will familiarize you with core CUDA concepts (kernels, threads, memory transfer, etc.) in bite-sized pieces [16]. Each sample demonstrates a specific API or optimization, so you can learn topic by topic.

- **CUB (CUDA Unbound)** – A "CUDA utility belt" library providing **reusable parallel primitives** (like parallel scan, sort, reductions, histogram, etc.) highly optimized for NVIDIA GPUs [17]. CUB abstracts many low-level details and offers templated routines that achieve near-peak hardware performance. By reading its implementations (now part of NVIDIA's unified Core Compute Libraries), you'll learn about techniques like warp-synchronization, prefix-sum algorithms on GPU, warp-aggregated atomics, and other tricks for **parallel algorithms** [17]. It's an excellent resource if you want to see how fundamental algorithms can be implemented efficiently on GPU hardware.

- **FlashAttention** – A cutting-edge CUDA extension for deep learning, implementing the "FlashAttention" algorithm for fast, memory-efficient Transformer attention. This open-source repo (by Tri Dao et al.) provides an optimized *exact attention* kernel that's used in many ML frameworks [18]. It's more specialized, but noteworthy: the code uses tiling and on-chip memory cleverly to avoid the usual $O(n^2)$ memory overhead of attention. Studying FlashAttention's CUDA

kernels (and perhaps its new **FlashAttention-2/3** improvements) can show you how to optimize algorithms that are not straight GEMMs. It's also a real-world example of a research idea (IO-aware attention) turned into production-quality GPU code [19].

- **Tiny-CUDA-NN** – A *"tiny CUDA neural network"* framework from NVIDIA Research (by Thomas Müller). It's a **lightweight C++/CUDA library for neural networks** that achieved fame for its *fully fused MLP* implementation (used in instant neural graphics, NeRF, etc.) [20]. The repo contains a highly optimized fully-fused **multi-layer perceptron** (where all layers' operations are fused into one CUDA kernel) and a fast multi-resolution hash encoding, among other components [21]. Because it's relatively small and self-contained, it's a fantastic codebase to read and emulate if you're interested in GPU acceleration for AI. You can see how the author uses CUDA **template metaprogramming and bit-level tricks** to fuse operations and minimize memory access, resulting in orders-of-magnitude speedups.

- **OpenAI Triton** – Not a repository of kernels per se, but a **GPU kernel programming framework** worth mentioning. *Triton is a Python-like DSL and compiler for writing custom GPU kernels with less effort than CUDA** [22]. Many researchers use Triton to craft high-performance kernels (including on AMD via recent ports). If you're open to alternatives, you might try writing some Triton kernels – its philosophy is to raise productivity by handling some low-level scheduling for you [22]. The skills carry over to CUDA, since you still think about memory tiling and parallelism, just at a slightly higher level of abstraction.

Each of these codebases is **high quality** and maintained by experts, so copying/typing out portions of them will expose you to *idiomatic, performance-conscious coding styles*. Pay attention to how they structure memory accesses, manage threads, and use C++ features (or PTX intrinsics/inline assembly in some cases) to push performance.

Tips to Optimize Your Code-Learning Workflow

Your approach of **typing out code from famous repositories** can be an effective way to learn—especially since you find you retain more by writing than by just reading. To get even more out of it, here are some tips and enhancements:

- **Combine Copying with Understanding:** When you sit down to re-type code, don't treat it as a pure transcription exercise. Take it slow enough to **absorb what each part of the code is doing**. For example, before typing a complex loop or CUDA kernel, read it and try to reason about its purpose (maybe even add your own comments in plain language). This ensures you're building understanding, not just muscle memory.

- **Run and Experiment:** Whenever possible, *build and run* the code you're copying. Seeing output or running tests will give context. If it's a library or algorithm, try writing a tiny driver (or use provided examples) to execute the code. This way, if you made a mistake while typing, you can debug it – which is another valuable skill. Also, try **small modifications** to the code after you've copied it. For instance, change a parameter, add a simple feature, or optimize something trivially. This helps test your comprehension and keeps the exercise engaging.

- **Segment Your Sessions:** For large codebases, break your typing sessions into logical chunks (perhaps one function or module at a time). After each chunk, reflect on it. You could even write a short summary of what you just typed (or explain it to a rubber duck, as the classic technique

goes). This reflection solidifies knowledge and highlights areas you're unsure about so you can revisit them.

- **Leverage Documentation & Comments:** Most high-quality repos come with READMEs, design docs, or inline comments (like the **comments in NVIDIA's samples or the Triton tutorials**). Make sure to read those alongside the code. The authors often explain the reasoning behind certain implementations or provide hints for understanding. It's like having a mentor guiding you through the code – take advantage of it.

- **Alternate Between Reading and Writing:** It can also help to sometimes read through a file or function entirely *before* typing it. Get the big picture, then attempt to write it out from scratch and compare to the original. This is a more active recall approach and can boost your understanding and memory. If you get it wrong, you'll learn from the correction.

- **Use Tools to Your Advantage:** When copying code, a plain code editor or a minimal online IDE might be ideal (to avoid too much auto-complete or linting that does the work for you). However, you can still use tools for learning: e.g., use an editor/IDE to navigate to function definitions, or use a debugger to step through the code you typed. These activities can reveal *dynamic behavior*, complementing the static typing practice.

In essence, keep doing what keeps you motivated (typing code for practice), but incorporate these habits to ensure it's **active learning**. The goal is not only to type faster but to internalize patterns and techniques from great code – so that when you write your own, you have a rich repertoire to draw from.

Expanding into New Domains – Projects to Consider Next

You mentioned interest in areas like drones, AI agents, reinforcement learning, web dev, and even low-level C/assembly. Diversifying your "code copying" across domains is a fantastic idea to broaden your skill set. Here are some **high-quality, interesting repositories** (and codebases) in various domains that you might enjoy exploring:

- **Drone Autopilot (C++):** *PX4 Autopilot* – This is a professional-grade open-source drone flight control system used in many UAVs. The PX4 codebase (mostly C++) covers real-time sensor data processing, control loops, state estimation, and mission logic. It's a large system, but you could start with specific modules (e.g. the attitude controller or the navigator). Reading PX4 will expose you to embedded C++ patterns and real-time system design. (Another simpler option is *ArduPilot* or even the firmware of a smaller drone like the Crazyflie, depending on your interest.) These projects show how C/C++ is used to interface with hardware and keep strict performance guarantees – very cool for a drone enthusiast.

- **Reinforcement Learning & Agents (Python):** *CleanRL* – A clean, minimalist RL library that provides **single-file implementations** of popular RL algorithms [23]. For example, CleanRL has a single Python file for PPO, one for DQN, etc., each under 500 lines but written in an easy-to-follow style [24]. Typing out a PPO implementation from CleanRL is a great way to learn the ins-and-outs of training loops, policy networks, and optimization logic without drowning in too much abstraction. Because each algorithm is self-contained, you'll see how the math translates directly to code. This can complement your interest in AI agents – after grasping the algorithms via CleanRL, you might look at more complex agent frameworks (like OpenAI Baselines or RLlib) with a better foundation.

- **Game Engines / Systems (C & some Assembly):** *Id Software's Quake Source* – The original Quake game engine (1996) is open-sourced in C with some hand-written x86 assembly for performance [25]. It's a legendary codebase that many game developers have learned from. By studying Quake, you'll see how a classic game loop, rendering pipeline, and memory management were implemented in an era of constrained hardware (and you'll encounter John Carmack's famed concise coding style). Notably, Quake's software renderer contains optimized assembly to rasterize graphics – you can try to read or copy some of that to get a feel for low-level optimization (though you can also compile Quake **without** the ASM for simplicity) [25]. If Quake feels too dated, you could look at **Godot Engine** (a modern open-source game engine in C++), but Godot is much larger. Quake is manageable and comes with the nostalgia factor, while also teaching you about data structures for 3D worlds, event handling, and more – all in straightforward C code.

- **Operating Systems / Low-Level (C/Assembly):** *MIT's xv6 OS* – **xv6** is a well-known teaching operating system, essentially a modern re-implementation of UNIX Version 6 for educational purposes [26]. It's written in **ANSI C with a small amount of x86 assembly**, and the entire source is compact (~10K lines). Copying xv6 (or just key pieces of it) can teach you a ton about how an OS works: process management, file systems, context switching (see `swtch.S` for the assembly context switch code), system call handling, etc. The code is clean and thoughtfully commented as it's meant for students [26]. If you're curious about how things like `fork()`, virtual memory, or device drivers look at a code level, xv6 is perfect. There's even an accompanying commentary/book (the "xv6 book") if you want explanations. Working through this will definitely flex your C and low-level skills.

- **Modern Web Development (JavaScript/TypeScript):** *Svelte* – A popular new web UI framework that takes a **compiler-centric approach** to building web apps. The Svelte repository is in TypeScript, and what's interesting is that Svelte is **not just a runtime library but a compiler that translates your declarative components into efficient JS** [27]. By reading Svelte's code, you'll see how a framework can differ from React/Vue: it handles reactivity at compile-time. The code involves parsing, AST transforms, and code generation – a different flavor of programming compared to GPU kernels, but high-quality and insightful for web. The tagline is that "Svelte is a new way to build web applications – a compiler that takes your declarative components and converts them into efficient JavaScript" [27]. Even if you don't copy the entire compiler, looking at parts of how Svelte updates the DOM or implements reactivity can broaden your perspective on front-end architecture.

- **(Bonus) New Systems Languages:** If you're up for something different, you might check out projects in languages like **Rust or Zig** which are gaining popularity for system-level programming. For example, *Bun* (an all-in-one JavaScript runtime written in Zig) or *Tokio* (an async runtime in Rust) are high-performance projects that could be fun to poke through. They aren't exactly what you mentioned, but since you enjoy systems and performance code, seeing a bit of Zig or Rust in action (with their strong safety guarantees) could be enlightening. This is more of a stretch goal, but worth mentioning!

Each of these suggestions comes with its own flavor and learning opportunities. As always, prefer **high-quality, well-documented repositories** (the ones above are known for that) and don't hesitate to read accompanying docs or research papers (for example, Svelte's website, or the CleanRL documentation) to supplement your understanding.

Finally, remember that **breadth is great, but depth is important too**. It's okay if you choose one or two projects from each category and really dive deep (rather than skimming many projects

superficially). By deeply engaging – i.e. coding by hand, running, modifying – you'll extract the most value and fun from each. Enjoy your "procrastilearning" journey, and happy coding! [2] [5]

---

[1] [2] [3] [4] [9] News – GPU MODE
https://www.gpumode.com/news

[5] [6] [7] GitHub - RadeonFlow/RadeonFlow_Kernels: Efficient implementation of DeepSeek Ops (Blockwise FP8 GEMM, MoE, and MLA) for AMD Instinct MI300X
https://github.com/RadeonFlow/RadeonFlow_Kernels

[8] GitHub - luongthecong123/fp8-quant-matmul: Block scaling for fp8 quantization matrix multiplication. Solution to GPU mode AMD challenge. Additionally, this repo includes codes for quantizing Pytorch bf16 matmul with fp8.
https://github.com/luongthecong123/fp8-quant-matmul

[10] GitHub - Snektron/gpumode-amd-fp8-mm: My submission for the GPUMODE/AMD fp8 mm challenge
https://github.com/Snektron/gpumode-amd-fp8-mm

[11] Deep Dive into Matrix Optimization on AMD GPUs
https://seb-v.github.io/optimization/update/2025/01/20/Fast-GPU-Matrix-multiplication.html

[12] [13] FP8 GEMM for AMD MI300x: Optimizations
https://akashkarnatak.github.io/amd-challenge/

[14] GitHub - NVIDIA/cutlass: CUDA Templates for Linear Algebra Subroutines
https://github.com/NVIDIA/cutlass

[15] [16] GitHub - NVIDIA/cuda-samples: Samples for CUDA Developers which demonstrates features in CUDA Toolkit
https://github.com/NVIDIA/cuda-samples

[17] CUB — CUDA Core Compute Libraries - GitHub Pages
https://nvidia.github.io/cccl/cub/

[18] [19] GitHub - Dao-AILab/flash-attention: Fast and memory-efficient exact attention
https://github.com/Dao-AILab/flash-attention

[20] [21] GitHub - NVlabs/tiny-cuda-nn: Lightning fast C++/CUDA neural network framework
https://github.com/NVlabs/tiny-cuda-nn

[22] GitHub - triton-lang/triton: Development repository for the Triton language and compiler
https://github.com/triton-lang/triton

[23] [24] vwxyzjn/cleanrl: High-quality single file implementation of ... - GitHub
https://github.com/vwxyzjn/cleanrl

[25] GitHub - id-Software/Quake: Quake GPL Source Release
https://github.com/id-Software/Quake

[26] GitHub - mit-pdos/xv6-public: xv6 OS
https://github.com/mit-pdos/xv6-public

[27] sveltejs/svelte: web development for the rest of us - GitHub
https://github.com/sveltejs/svelte