Linux® Support for CPU Stress Test on Large-Scale AMD EPYC™ Systems

S. Biplab Raut
AMD India Private Limited, Bangalore, India
biplab.raut@amd.com

***Abstract*—The AOCL CPU Stress Test (CST) validates processor stability under sustained computational and memory pressure using AMD's optimized BLIS and LibFLAME numerical libraries. Originally developed for Windows desktop environments with a 64-core limit, CST failed critically when deployed on high core count AMD server-class CPU platforms, such as a dual-socket EPYC™ 9654 system with 192 physical cores and 1.5 TB DDR5: CPU stress achieved only 2% utilization, memory setup timed out, and blend mode produced 0% CPU activity. This paper presents four engineering solutions that enable CST on Linux at server scale: a POSIX® threading and synchronization layer, mmap-based parallel memory allocation with transparent huge pages, an L3-cache-topology-aware two-level OpenMP threading model, and a restructured synchronization protocol for blend mode. The resulting system achieves 99% CPU utilization across all 192 cores, 1,403 GB (95%) memory residency with setup in approximately 5 seconds, and simultaneous CPU-plus-memory stress – all with clean cooperative shutdown within 60 seconds.**

***Index Terms*—AMD EPYC, AOCL Numerical Libraries, AVFS, BLIS, CPU Stress Testing, LibFLAME, Linux, OpenMP**

I. Introduction

High-performance CPU platforms like the ones used for gaming or infotainment require rigorous stability validation, particularly after the voltage-frequency tuning done for achieving optimal CPU performance. Overclocking increases clock speeds beyond factory specifications, while undervolting reduces voltage to lower power. Both techniques modify the adaptive voltage and frequency scaling (AVFS) curve [1] – the voltage-frequency characteristic of a processor. While scaling the voltage and frequency adaptively beyond the default factory settings, the CPU may enter the unstable operating region leading to hardware exceptions and system crashes. Therefore, the CPU's stability needs to be continuously monitored and must be validated under sustained heavy load to ensure system stability. CPU Stress Test tools as illustrated in Fig. 1 are used to stress the CPU system operating at a given power-performance power state model (PSM) offset point on the AVFS curve for validating the performance, stability, and reliability of the CPU under heavy load.

The AOCL CPU Stress Test (AOCL-CST) [21] tool stresses AMD processors and other x86 CPUs as well through computationally intensive workloads drawn from the AMD Optimizing CPU Libraries (AOCL) [2]. CST employs BLIS [3] [20] for general matrix-matrix multiplication (GEMM), providing memory-bandwidth-intensive stress, and LibFLAME [4] for the LAPACK divide-and-conquer SVD driver *gesdd, providing compute-intensive stress. CST operates in three modes: Mode 0 (CPU), Mode 1 (memory), and Mode 2 (blend). Fig. 2 summarizes the modular CST flow.

©2026 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, Ryzen and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

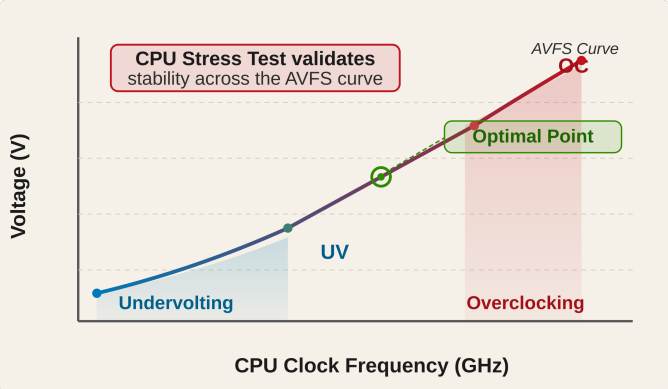


Fig. 1. AVFS curve and CPU stress test validation scope.

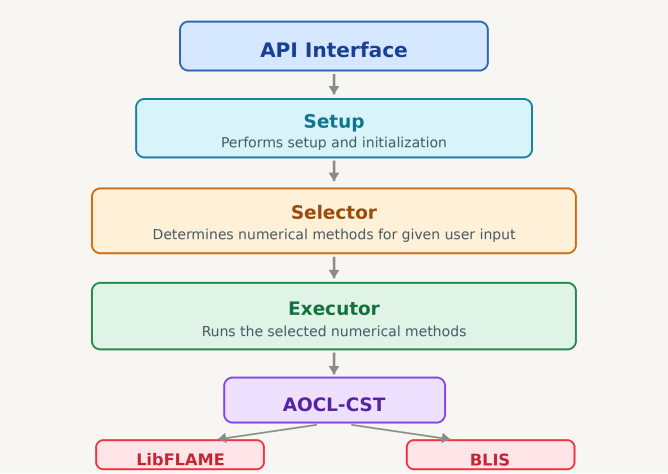


Fig. 2. AOCL-CST modular architecture.

CST was originally developed for Windows desktop and workstation environments, where the execution model, thread management, and CPU affinity handling were shaped by Win32 APIs and the 64-core processor group abstraction. This design worked well for desktop-class use cases such as Ryzen™ Master [5] [6] [7], but it did not naturally scale to modern Linux-based [19] AMD server platforms, where

high core counts, large NUMA memory capacity, and CCD/L3 cache topology introduce very different stress-test requirements. This paper presents the porting and optimization of CST for Linux, targeting large-scale AMD CPU systems with 128 to 256 cores and memory capacity beyond 1 TB, while preserving the original Windows functionality and enabling responsive, full-system CPU and memory stress execution at server scale.

The key contributions of this work are: (1) a complete Linux-compatible POSIX threading and synchronization layer while preserving the existing Windows execution path, (2) a five-point memory optimization for systems with more than 1 TB of memory, (3) an L3-topology-aware two-level OpenMP [18] model achieving high CPU utilization across high-core-count AMD server CPUs, and (4) a redesigned hybrid mode synchronization to run without any stalls.

II. Problem Statement

A. Background

CST was designed for Windows desktop and workstation environments, where the execution model was built around Win32 threading, event signaling, processor group affinity, and standard heap-based memory allocation. This design was suitable for desktop-class use cases, including AVFS curve optimizer used by tools such as AMD Ryzen Master.

Linux-based AMD server platforms introduce a very different validation environment. Modern AMD CPU systems can expose hundreds of logical cores, exceeding the 64-core processor group assumptions used in the original Windows implementation. These systems also often provide memory capacity beyond 1 TB, making serial memory initialization and repeated full-buffer copies impractical within short stress-test windows. In addition, multi-CCD server CPUs require topology-aware scheduling so that CPU stress threads align with shared L3 cache groups and avoid cross-node bandwidth bottlenecks. A validation stress tool must therefore start quickly, sustain high CPU and memory pressure, and still support clean exit behavior when invoked by automation frameworks.

B. Gap Analysis and Challenges

Moving CST from a Windows desktop execution model to Linux server-class systems exposed gaps at multiple layers of the implementation. Some issues were direct portability gaps, such as Win32 thread events, dynamic loading, and processor-group affinity assumptions. Others were scale-related: memory setup that was acceptable on desktop systems became impractical beyond 1 TB, and a naive expansion of numerical workloads across hundreds of cores created nested-thread oversubscription. Table I summarizes the main gaps and their technical impact.

TABLE I
GAP ANALYSIS FOR LINUX SERVER-SCALE CST

Area	Gap	Technical impact
Portability	Win32 events, thread handles, and dynamic loading APIs made it Windows specific implementation.	Required POSIX synchronization, pthread-safe type handling, and Linux shared-library loading.
Affinity	Processor-group logic assumed 64-core Windows groups.	Rejected or misrepresented large Linux CPU ranges with hundreds of logical cores.
Memory setup	Serial first-touch and redundant full-buffer copies scaled poorly beyond 1 TB.	Setup time exceeded the stress window before useful memory pressure could begin.
NUMA placement	Heap allocation with serial first-touch could place large buffers on limited NUMA nodes.	Remote memory access created bandwidth bottlenecks and reduced effective memory-stress quality.
CPU stress	Serial GESDD underutilized cores; naive parallel expansion created thread oversubscription.	Caused low CPU utilization, thread explosion, and hangs at high core counts.
Blend mode	CPU stress was gated behind long memory-copy phases.	Hybrid mode reported memory activity but failed to start CPU stress promptly.

C. Previous Work

Existing CPU stress tools [8]–[12] such as Prime95 [15], AIDA64 [16], OCCT [13], RealBench [14], and y-cruncher [17] are widely used for general stability testing and overclocking workflows. These tools are effective at creating heavy load, but they are generally workload-centric rather than topology-aware: they do not explicitly map numerical stress kernels to AMD CCD/L3 cache groups, nor do they use AMD-optimized BLIS and LibFLAME execution paths as validation workloads.

The earlier Windows version of CST addressed a different problem space: desktop-class AVFS validation, including Ryzen Master usage, where the operating system, core-count envelope, and memory footprint were substantially smaller. Large Linux server systems introduce a different class of constraints. Stress must ramp quickly, cover high-core-count CPUs, operate over memory capacities beyond 1 TB, respect NUMA and chiplet topology, and remain responsive to automation frameworks. Prior tools and the original CST implementation did not address these requirements as an integrated Linux server-scale design.

III. Proposed Solution

We reworked CST into a Linux-compatible, large-scale CPU stress test while preserving the existing Windows behavior. The solution is organized around four practical areas: platform features, memory and setup optimizations, topology-aware execution design, and correctness fixes (Fig. 3). We kept the Windows path isolated behind conditional compilation and introduced Linux-specific mechanisms only where the operating system, memory model, or topology demanded a different implementation.

A. Platform Porting (Features)

We first replaced the Windows-only execution layer with Linux equivalents while keeping the CST control flow familiar. Win32 events were mapped to a POSIX event object implemented with a mutex, condition variable, and auto-reset flag. `SetEvent` sets the flag and signals the condition variable; `WaitForSingleObject_Linux` blocks with optional timeout and clears the flag when it consumes the

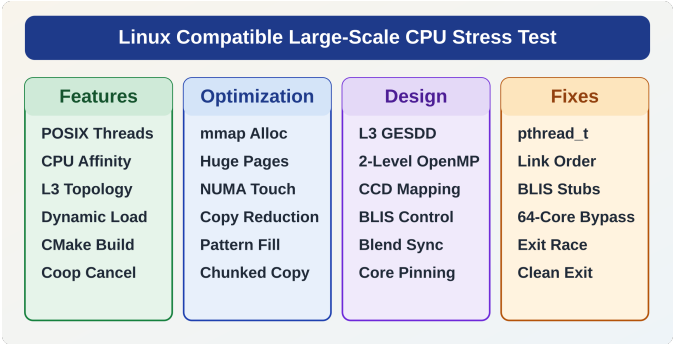


Fig. 3. Linux-compatible large-scale CST solution categories.

event. Separately, `AOCL_BLIS_should_exit()` peeks at the per-thread exit event without resetting it, which allows long memory loops to exit cooperatively. CPU affinity moved from Windows processor groups to `sched_setaffinity`, and Linux dynamic loading uses `dlopen`/`dlsym`. We also kept the Linux additions behind platform guards so the Windows path remains unchanged.

$$\begin{aligned} \text{Event} &= \{\text{mutex}, \text{condvar}, \text{flag}\} \\ \text{SetEvent} &: \text{flag} = 1,\ \text{signal}(\text{condvar}) \\ \text{Wait} &: \text{block until flag} = 1,\ \text{then flag} = 0 \end{aligned}$$

B. Memory Optimization

We then redesigned the memory setup path because the original sequence spent most of the test window preparing matrices instead of stressing memory. The Linux path applies five changes. First, `mmap(MAP_ANONYMOUS)` allocates the large BLIS buffers and `madvise(MADV_HUGEPAGE)` is issued before any first touch, so the kernel can back the initial faults with transparent huge pages. Second, a single OpenMP [18] `schedule(static)` first-touch loop commits all four buffers in parallel, which spreads pages according to the executing threads instead of concentrating them on one NUMA node. Third, the serial `randm + norm + scale` path is replaced with a parallel fractional fill of 0.5; this avoids the all-ones special case while still keeping GEMM verification meaningful. Fourth, redundant `memcpy` passes are removed, leaving one controlled copy path from `C` to `A`, `B`, and `C_save`. Fifth, `CST_COPYM` performs chunked copies with exit checks, so a timer signal can stop setup cleanly instead of waiting for a large copy to finish.

$$\begin{aligned} W_{\text{before}} &= 10S \\ W_{\text{after}} &= 4S = \text{fill}(C) + \text{copy}(C, A, B, C_s) \end{aligned}$$

Here, S is one matrix footprint. The equation captures the dominant logical initialization and copy sweeps; the separate first-touch phase is intentionally parallel and NUMA-aware. This is why the setup becomes practical at terabyte scale without lowering the requested memory-stress coverage.

TABLE II
CPU STRESS THREADING APPROACHES ON EPYC 9654

Approach	Threads	CPU	Outcome
Serial GESDD (BLIS=192)	192	2%	Bandwidth bound
192 flat GESDDs	192+	99%	Hang at 96+ cores
24 CCDs × 8-thread BLIS	2,187	20%	Contention
24 CCDs × 8 independent GESDDs	192	99%	PASS

C. CPU Stress Architecture

For CPU stress, we avoided the two extremes that failed in practice: one large serial GESDD that used only a small fraction of the machine, and a naive flat parallel launch that oversubscribed BLIS, OpenMP, and LibFLAME threads. Instead, we map work to the hardware topology. The outer level follows shared L3/CCD groups, and the inner level runs one single-threaded GESDD worker per selected core. This keeps the stress compute-heavy, limits nested threading, and preserves cache locality. Table II compares the evaluated threading approaches.

The implementation detects L3 groups from /sys/devices/system/cpu/cpuN/cache/index3/shared_cpu_list [19], filters the groups to the selected core range, and pins inner workers with `sched_setaffinity`. The GESDD matrix dimensions are not hard-coded; they are derived from the detected L3 size using the same working-set model used in the code. We use complex double data, so each element contributes 16 bytes, and the default aspect ratio is $N \approx 1.3M$. BLIS is forced to one thread inside each GESDD worker, preventing BLIS/OpenMP/LibFLAME from multiplying the thread count.

$$\begin{aligned} W_{\text{GESDD}} &= 16(MN + M^2 + N^2 + \min(M, N)) \\ W_{\text{GESDD}} &\le \rho L3 \\ T_{\text{workers}} &= G_{L3} \times C_{L3}, \quad \text{BLIS}_{\text{threads}} = 1 \end{aligned}$$

Here, ρ selects the L3-fill tier, G_{L3} is the number of active shared-L3 groups, and C_{L3} is the number of selected cores inside each group. On the evaluated system this maps naturally to 24 groups with 8 workers each, but the formula remains topology-driven rather than platform-hard-coded.

D. Blend Mode and Critical Fixes

Finally, we fixed the blend-mode sequencing so CPU stress is not blocked by the entire memory setup path. In Mode 2, the memory side only needs to allocate the requested memory and keep it resident; it does not need to finish every matrix copy before CPU stress can start. The original flag `g_gemm_setup_complete` was reached after expensive fill and copy stages, so GESDD could wait for the full time limit and report 0% CPU. We moved the flag immediately after parallel first-touch, when all pages are already committed, and then the memory side enters a 500 ms hold loop with periodic checks. We also fixed `pthread_t` type safety, LibFLAME-before-BLIS link ordering, BLIS stub symbols for missing RM entry points, Linux bypass of the 64-core processor-group check, and mutex-protected exit signaling.

$$\text{old} : \text{ready after}\{FT, \text{fill}, \text{copies}\}$$
$$\text{new} : \text{ready after}\{FT\}, \quad \text{hold memory} \parallel \text{GESDD}_{L3}$$

In this sequence, FT denotes the parallel first-touch phase. The change aligns the synchronization point with the actual memory-residency requirement rather than with the later data-copy completion point.

IV. Experimental Results

All experiments were conducted on a dual-socket AMD EPYC 9654 system with 192 physical cores, 24 CCDs × 8 cores, 1.5 TB DDR5, RHEL 9.5, and a 60-second stress window. We evaluated the three CST execution modes independently and then verified scaling across selected core counts. The results show whether each Linux design point solves its target failure mode: CPU saturation, terabyte-scale memory residency, simultaneous blend stress, and core-count scaling.

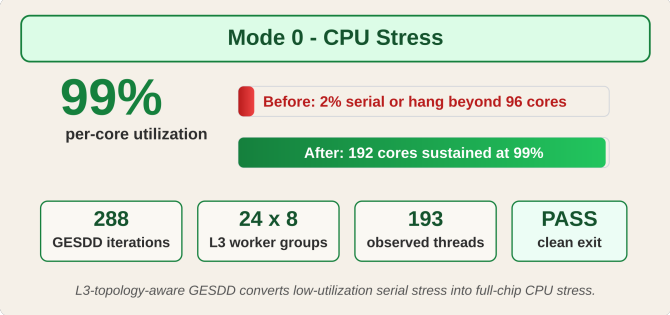


Fig. 4. Mode 0 CPU stress result.

Mode 0 demonstrates the CPU-side redesign (Fig. 4). The original serial GESDD path either used only a small fraction of the machine or failed when naively expanded beyond 96 cores. With L3-aware grouping, CST sustains 99% utilization across all 192 cores and completes 288 GESDD iterations. The important result is not just the high CPU percentage, but that the workload remains compute-heavy while avoiding nested BLIS/OpenMP/LibFLAME thread explosion.

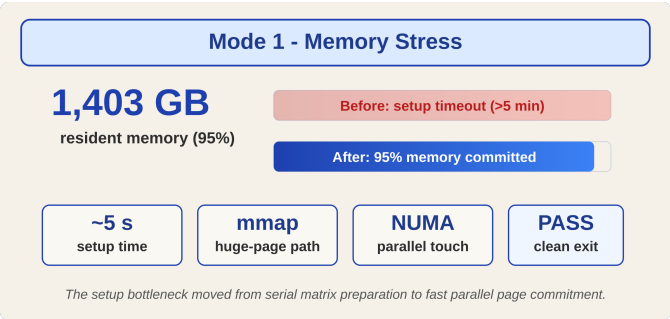


Fig. 5. Mode 1 memory stress result.

Mode 1 demonstrates the memory setup redesign (Fig. 5). The previous flow could spend the entire stress window in serial initialization and redundant copy passes, never reaching useful memory pressure. The Linux path commits 1,403 GB, approximately 95% of Linux-reported usable memory, with setup completing in about 5 seconds. This confirms that `mmap`, huge-page advice before first touch, NUMA-aware OpenMP first-touch, pattern fill, and chunked copies together remove the server-scale setup bottleneck without reducing the requested memory footprint.

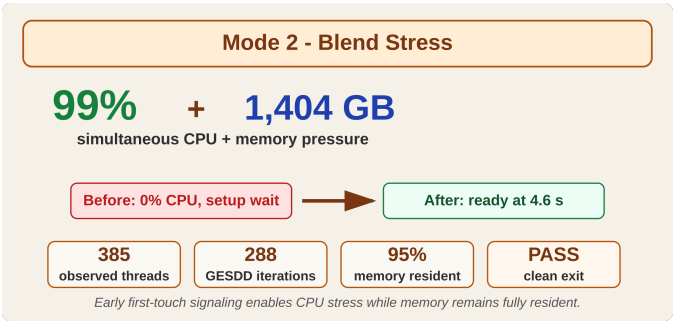


Fig. 6. Mode 2 blend stress result.

Mode 2 demonstrates the blend-mode synchronization fix (Fig. 6). Before the change, the CPU thread waited behind memory-copy phases and reported 0% CPU even though memory had already been committed. After moving the ready signal to the end of parallel first-touch, CST delivers 99% CPU utilization while holding 1,404 GB resident. The 4.6-second first-touch time is the key timing point: it is early enough for CPU stress to run for the rest of the stress window, yet late enough to guarantee that the memory pressure is real.

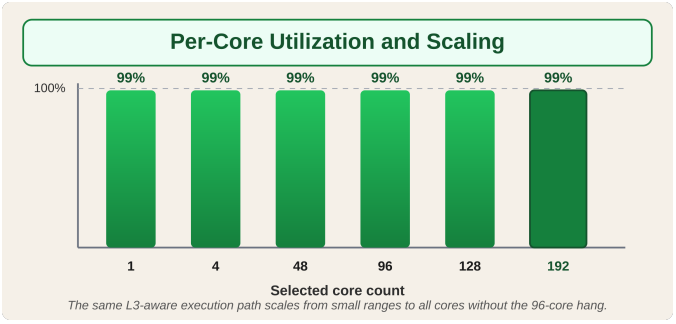


Fig. 7. Per-core utilization scaling from selected core ranges to all cores.

The scaling result (Fig. 7) shows that the same execution model holds from small core selections to the full 192-core system. Per-core utilization remains 99% for 1, 4, 48, 96, 128, and 192 selected cores, with clean exit at each scale. This is the main portability insight: the Linux CST path is no longer tied to the Windows 64-core group assumption or to a single fixed topology. It uses the detected L3 layout and the requested core range to shape the stress workload.

Overall, the results turn CST from a desktop-oriented Windows stress path into a Linux server-scale validation workload. Mode 0 demonstrates full-chip compute stress, Mode 1 demonstrates fast terabyte-scale memory residency, Mode 2 demonstrates simultaneous CPU-plus-memory pressure, and

the scaling result shows that the design can cover both partial-core and all-core validation runs. These properties are the practical requirements for using CST in automated EPYC silicon-validation stress flows.

V. Conclusion and Future Work

This work transforms AOCL-CST from a Windows desktop-oriented stress utility into a Linux-capable validation workload for large AMD EPYC server systems, where Linux is the dominant deployment environment and where AVFS and post-silicon validation require full-platform stress coverage. The key technical outcome is not only functional portability but also scalable stress fidelity: CST now sustains 99% CPU utilization across 192 cores, commits approximately 1.4 TB of memory within a short setup window, and runs blend mode with simultaneous CPU pressure and high memory residency. These results were enabled by four coordinated design changes: a POSIX synchronization and cooperative-cancellation layer, a NUMA-aware memory setup path using `mmap`, huge-page advice, parallel first-touch, and copy reduction, an L3-topology-aware GESDD execution model, and an early blend-mode synchronization point aligned with actual memory residency.

The broader impact is that CST can now exercise the same class of Linux server platforms where EPYC validation, AVFS characterization, and production qualification matter most. By using AOCL BLIS and LibFLAME numerical kernels as the stress payloads, the tool also provides a practical stability signal for the optimized library execution paths under extreme CPU, memory, and blended pressure. The L3-topology-aware design generalizes beyond one fixed core count because it derives work placement from the detected cache topology and selected core range. The implementation preserves the Windows path while adding Linux mechanisms that respect sockets, NUMA placement, CCD/L3 topology, and automation-driven clean exit requirements. The resulting Linux capability also positions CST as a deployable stress workload for silicon-validation workflows, including post-silicon stability testing and production processor qualification. Future work should focus on porting `CST_THREAD_SIGNALING` into the Linux BLIS build for interruptible GEMM, extending topology parsing for SMT and non-contiguous CPU identifiers, and using BLIS `rntm_t` for per-invocation thread control in more advanced concurrent GEMM/GESDD scenarios.

References

[1] AMD, “AMD Ryzen Master Utility for Overclocking Control,” AMD Technical Documentation, 2024.

[2] AMD, “AMD Optimizing CPU Libraries (AOCL),” https://www.amd.com/en/developer/aocl.html.

[3] F. G. Van Zee and R. A. van de Geijn, “BLIS: A Framework for Rapidly Instantiating BLAS Functionality,” *ACM Trans. Math. Software*, vol. 41, no. 3, pp. 1–33, 2015.

[4] AMD, “AMD Optimizing CPU Libraries – LibFLAME,” https://www.amd.com/en/developer/aocl/dense-linear-algebra-library.html.

[5] ”Ryzen Master”. Ryzen Master release page [online]. Available: https://www.amd.com/en/resources/support-articles/release-notes/RN-RYZEN-MASTER-2-14-2-3341.html

[6] ”Curve Optimizer”, Curve Optimizer Feature in Ryzen Master FAQ Page. [online]. Available: https://www.amd.com/content/dam/amd/en/documents/products/software-tools/faq-curve-optimizer.pdf

[7] ”Ryzen Master”, Ryzen Master quick reference guide. [online]. Available: https://www.amd.com/content/dam/amd/en/documents/products/software-tools/ryzen-master-quick-reference-guide.pdf

[8] ”Stress testing”. Stress testing categories. [online]. Available: https://wiki.archlinux.org/title/Stress_testing#Stress_testing_tasks

[9] ”IntelBurnTest”. Intel Burn Test. [online]. Available: https://www.techpowerup.com/download/intelburntest/

[10] ”LINPACK”. Netlib Linpack. [online]. Available: https://www.netlib.org/linpack/

[11] ”HandBrake”. HandBrake Open source video transcoder. [online]. Available: https://handbrake.fr/

[12] ”Cinebench”. Cinebench Homepage. [online]. Available: https://www.maxon.net/en/cinebench

[13] ”OCCT Stability testing”. OCCT Homepage. [online]. Available: https://www.ocbase.com/

[14] ”RealBench”. Asus Real Bench Homepage. [online]. Available: https://rog.asus.com/tag/real-bench/

[15] ”Prime95 Stresstest”. Prime95 Download page [online]. Available: https://www.mersenne.org/download/#stresstest

[16] ”AIDA64”. AIDA64 Home Page. [online]. Available: https://www.aida64.com/user-manual/tools/system-stability-test

[17] ”y-cruncher”. y-cruncher - A Multi-Threaded Pi Program. [online]. Available: https://www.numberworld.org/y-cruncher/

[18] OpenMP Architecture Review Board, “OpenMP Application Programming Interface, Version 5.0,” Nov. 2018.

[19] Linux Kernel Documentation, “CPU Topology – sysfs Interface,” https://www.kernel.org/doc/html/latest/admin-guide/cputopology.html

[20] ”BLIS”. Blis Github Home page [online]. Available: https://github.com/amd/blis

[21] S. B. Raut, ”AOCL-CST - A new CPU Stress Test library for AMD CPUs,” 2025 IEEE High Performance Extreme Computing Conference (HPEC), USA, 2025, pp. 1-4