GPU-Accelerated Pattern Matching for Open Anti-Virus Engines

A Thesis

submitted to the designated

by the Assembly

of the Department of Computer Science and Engineering

Examination Committee

by

Vasileios Kolokythas

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN DATA AND COMPUTER SYSTEMS ENGINEERING

WITH SPECIALIZATION

IN ADVANCED COMPUTER SYSTEMS

University of Ioannina

School of Engineering

Ioannina 2026

Examining Committee

- Christos Liaskos, Associate Professor, Department of Computer Science and Engineering, University of Ioannina (Advisor)
- Evangelos Papapetrou, Associate Professor, Department of Computer Science and Engineering, University of Ioannina
- Sotirios Ioannidis, Professor, School of Electrical and Computer Engineering, University of Crete

Acknowledgements

I would like to begin by offering my sincere thanks to my supervisor, Assistant Professor Mr. Christos Liaskos of the University of Ioannina, for his invaluable mentorship and constant encouragement throughout this journey. His profound expertise and passion for innovation have been a major source of inspiration, sparking my initial interest in this research domain. I am incredibly grateful for his patience, his belief in my potential, and the endless discussions that challenged me to evolve into a more rigorous and independent scientist. I extend my heartfelt appreciation to Dr. Eva Papadogiannaki, PhD researcher from the University of Crete, Department of Computer Science. Her deep knowledge, critical feedback, and hands-on guidance significantly enhanced the quality of this thesis and were crucial to the successful realization of this architecture. My heartfelt thanks also go to my family, whose continuous emotional and financial support made this achievement possible.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

The rapid proliferation and increasing architectural complexity of contemporary malware have introduced severe computational strains on signature-based virus scanning infrastructures, exposing deep processing bottlenecks within legacy, CPU-bound sequential matching engines. Traditional implementations of the Aho–Corasick pattern-matching algorithm rely on recursive, pointer-chasing traversals between automaton nodes, producing poor data locality, frequent cache invalidations, and strict single-threaded execution limits that prevent real-time filesystem auditing at scale.

To overcome these structural constraints, this thesis introduces a highly optimized, massively parallel pattern-matching architecture natively integrated into the production codebase of ClamAV 1.0.0 — the industry-standard open-source antivirus engine maintained by Cisco Talos Intelligence Group and globally deployed within high-volume network gateways and cloud storage perimeters. The developed system is built around three core engineering contributions.

First, ClamAV's default internal representation — a recursive, pointer-linked sparse trie — is replaced with a fully linearized Deterministic Finite Automaton (DFA) stored as a flat two-dimensional state transition table. This restructuring eliminates pointer-chasing traversal and replaces it with a constant-time array lookup mapping any (state, byte) pair to a next state. Second, this flat table representation compresses the full ClamAV signature database of 3,968,449 signatures — with the dominant matcher encoding 2,651,902 patterns across 486,426 unique states — into a space-optimized 475 MB transition table that is loaded directly onto the GPU via OpenCL, residing entirely within device memory. Third, a hybrid dispatch layer manages work partitioning between the GPU and the host CPU. Files exceeding a measured 256 KB crossover threshold are transferred to the GPU asynchronously; patterns structurally incompatible with parallel evaluation are redirected to CPU threads, preserving correctness across the full signature set.

The system was evaluated against three independent live malware corpora spanning ten file type categories, using the full production ClamAV 1.0.0 signature database of 3,968,449 signatures. Each configuration was measured over $n = 10$ repeated runs with 95% confidence intervals computed via Student's t-distribution ($t_{0.025,9} = 2.262$). Peak speedup over the single-threaded CPU baseline reached $15.13\times$ on VirusShare Corpus 2 ELF binaries (119 files, 173.43 MB), with consistent speedups observed across all file types whose mean size exceeds the empirically determined 256 KB dispatch threshold. Detection accuracy against the CPU ground truth yielded 95.20% recall, 100% precision, and an F_1 score of 0.9754 across 3,837 ground-truth positive files. Zero false positives were observed in any partition across any corpus.

Detection accuracy analysis against the CPU ground truth yields an overall **recall of 95.20%**, **precision of 100%**, and **$\mathbf{F_1}$ score of 0.9754**. The 4.80% missed-detection rate is entirely attributable to an architectural constraint — only one of ClamAV's multiple Aho–Corasick matcher roots is currently loaded onto the GPU — and not to any error in the pattern-matching logic itself. The GPU engine produces zero false positives against the CPU ground truth on any file type where it evaluates signatures from its loaded root. Furthermore, shifting the dominant pattern-matching load to the graphics accelerator reduces host CPU utilization during active scans, freeing critical processor capacity for concurrent system tasks. This research presents a cloud-native, open-source framework for hardware-accelerated malware detection suitable for deployment in hyperscale cloud environments and high-throughput network security perimeters.

Εκτεταμενη Περιληψη

Η αδιάκοπη εξάπλωση και η αρχιτεκτονική πολυπλοκότητα του σύγχρονου κακόβουλου λογισμικού έχουν επιφέρει πρωτοφανή υπολογιστική επιβάρυνση στις υποδομές ελέγχου υπογραφών, αναδεικνύοντας σοβαρά προβλήματα συμφόρησης στις παραδοσιακές σειριακές μηχανές επεξεργασίας που βασίζονται αποκλειστικά στην CPU. Οι συμβατικές υλοποιήσεις του αλγορίθμου αντιστοίχισης προτύπων Aho–Corasick βασίζονται σε αναδρομικές δομές δεικτών, οι οποίες χαρακτηρίζονται από κακή τοπικότητα δεδομένων, συχνές ακυρώσεις κρυφής μνήμης και περιορισμούς μονονηματικής εκτέλεσης που εμποδίζουν τον πραγματικό χρόνο έλεγχο του συστήματος αρχείων σε μεγάλη κλίμακα.

Για την άρση αυτών των δομικών περιορισμών, η παρούσα διατριβή εισάγει μια εξαιρετικά βελτιστοποιημένη, μαζικά παράλληλη αρχιτεκτονική αντιστοίχισης προτύπων, πλήρως ενσωματωμένη στον κώδικα παραγωγής της μηχανής ClamAV 1.0.0. Το ClamAV, το οποίο συντηρείται από την Cisco Talos Intelligence Group, αποτελεί την παγκόσμια βιομηχανική σταθερά στις ανοιχτού κώδικα μηχανές ανίχνευσης κακόβουλου λογισμικού, με ευρεία ανάπτυξη σε πύλες δικτύου υψηλού φόρτου και συστήματα αποθήκευσης cloud. Το σύστημα που αναπτύχθηκε βασίζεται σε τρεις βασικές μηχανολογικές συνεισφορές.

Πρώτον, η εγγενής εσωτερική αναπαράσταση του ClamAV — ένα αναδρομικό, αραιό δέντρο με δείκτες — αντικαθίσταται από έναν πλήρως γραμμικοποιημένο Ντετερμινιστικό Πεπερασμένο Αυτόματο (DFA) που αποθηκεύεται ως επίπεδος δισδιάστατος πίνακας μεταβάσεων. Αυτή η αναδιάρθρωση εξαλείφει την αναδρομική διάσχιση μέσω δεικτών και την αντικαθιστά με μια αναζήτηση πίνακα σταθερού χρόνου που αντιστοιχίζει κάθε ζεύγος (κατάσταση, byte) σε επόμενη κατάσταση. Δεύτερον, αυτή η επίπεδη αναπαράσταση συμπυκνώνει τη συνολική βάση δεδομένων υπογραφών ClamAV των 3.968.449 υπογραφών — με τον κύριο αντιστοιχιστή να κωδικοποιεί 2.651.902 πρότυπα σε 486.426 μοναδικές καταστάσεις — σε έναν

βελτιστοποιημένο πίνακα μεταβάσεων 475 MB, ο οποίος φορτώνεται απευθείας στη GPU μέσω OpenCL, διαμένοντας εξ ολοκλήρου στη μνήμη της συσκευής. Τρίτον, ένα επίπεδο υβριδικής κατανομής διαχειρίζεται τον διαμερισμό εργασίας μεταξύ GPU και CPU. Αρχεία που υπερβαίνουν το εμπειρικά προσδιορισμένο όριο των 256 KB μεταφέρονται ασύγχρονα στη GPU· πρότυπα που δεν είναι συμβατά με παράλληλη αξιολόγηση ανακατευθύνονται σε νήματα CPU, διατηρώντας την ορθότητα σε ολόκληρο το σύνολο υπογραφών.

Το σύστημα αξιολογήθηκε σε τρία ανεξάρτητα σύνολα δεδομένων κακόβουλου λογισμικού, που καλύπτουν δέκα κατηγορίες τύπων αρχείων, χρησιμοποιώντας την πλήρη βάση δεδομένων υπογραφών του ClamAV 1.0.0 με 3.968.449 υπογραφές. Κάθε διαμόρφωση μετρήθηκε σε $n = 10$ επαναλαμβανόμενες εκτελέσεις με διαστήματα εμπιστοσύνης 95% υπολογισμένα μέσω της t-κατανομής του Student ($t_{0,025,9} = 2{,}262$). Η μέγιστη επιτάχυνση έναντι της μονής γραμμής βάσης CPU έφτασε $15{,}13\times$ στα δυαδικά αρχεία ELF του VirusShare Corpus 2 (119 αρχεία, 173,43 MB), με σταθερές επιταχύνσεις σε όλους τους τύπους αρχείων των οποίων το μέσο μέγεθος υπερβαίνει το εμπειρικά προσδιορισμένο όριο κατανομής των 256 KB. Η ακρίβεια ανίχνευσης έναντι της γραμμής βάσης CPU απέδωσε ανάκληση 95,20%, ακρίβεια 100% και βαθμολογία F_1 ίση με 0,9754 σε 3.837 θετικά αρχεία αναφοράς. Δεν παρατηρήθηκαν ψευδώς θετικά αποτελέσματα σε κανένα υποσύνολο κανενός συνόλου δεδομένων.

Η ανάλυση ακρίβειας ανίχνευσης σε σχέση με τον έλεγχο αλήθειας της CPU αποδίδει συνολική **ανάκληση 95,20%**, **ακρίβεια 100%** και **βαθμολογία $\mathbf{F_1}$ 0,9754**. Το ποσοστό αστοχίας ανίχνευσης 4,80% οφείλεται αποκλειστικά σε αρχιτεκτονικό περιορισμό — μόνο μία από τις πολλαπλές ρίζες αντιστοιχιστή Aho–Corasick του ClamAV φορτώνεται επί του παρόντος στη GPU — και όχι σε κανένα σφάλμα στη λογική αντιστοίχισης προτύπων. Η μηχανή GPU δεν παράγει ψευδώς θετικά αποτελέσματα έναντι του ελέγχου αλήθειας της CPU σε κανέναν τύπο αρχείου όπου αξιολογεί υπογραφές από τη φορτωμένη ρίζα της. Επιπλέον, η μεταφορά του κυρίαρχου φόρτου αντιστοίχισης προτύπων στον επιταχυντή γραφικών μειώνει τη χρήση της host CPU κατά τη διάρκεια ενεργών σαρώσεων, απελευθερώνοντας κρίσιμη υπολογιστική ισχύ για ταυτόχρονες εργασίες του συστήματος. Η παρούσα έρευνα παρέχει ένα ανοιχτού κώδικα, cloud-native πλαίσιο για επιταχυνόμενη ανίχνευση κακόβουλου λογισμικού, κατάλληλο για ανάπτυξη σε υποδομές cloud υπερκλίμακας και υψηλής απόδοσης περιμέτρους ασφάλειας δικτύου.

Chapter 1

Introduction

1.1 Problem Statement

The volume and architectural complexity of malware have grown to a point where traditional antivirus scanning infrastructure can no longer keep pace. The AV-TEST Institute has documented over 1.5 billion known malicious samples, and the rate of new discoveries continues to accelerate. At the same time, the environments in which antivirus engines must operate have become increasingly demanding: enterprise mail servers process thousands of file attachments per day, cloud storage platforms require real-time inspection of uploaded content, web gateways must screen high-throughput HTTP traffic, and endpoint devices impose strict resource constraints.

The core of the problem is algorithmic. Signature-based antivirus engines identify malware by scanning files for known byte sequences drawn from a database of threat signatures. The standard algorithm used for this task, Aho–Corasick, is inherently sequential: it processes one byte at a time, following a chain of state transitions through an automaton built from the signature database. On a single CPU core, a full scan of a large file can take hundreds of milliseconds. Across millions of files per day, this latency makes real-time, line-rate protection fundamentally unachievable under a sequential execution model.

1.2 The Gap: Unexploited GPU Parallelism

Graphics Processing Units (GPUs) are now ubiquitous across the full spectrum of modern computing, from desktop workstations to large-scale cloud infrastructure. Their massively parallel architecture, designed to apply the same operation to thousands of data elements simultaneously, is in principle well suited to the repetitive, data-parallel nature of pattern matching. Yet this potential has not been realized in practice for antivirus scanning. Four structural obstacles account for this gap.

First, the standard Aho–Corasick implementation relies on a pointer-linked trie, a data structure whose irregular, data-dependent memory access patterns are fundamentally incompatible with the lockstep execution model of GPU hardware. Second, compiling millions of threat signatures into a single Deterministic Finite Automaton (DFA) produces a state space so large that the resulting data structure historically exceeds the capacity of GPU video memory, forcing repeated access to slower off-chip VRAM. Third, the fixed overhead of transferring data to the GPU over the PCIe bus negates any throughput advantage for files below a certain size, requiring a carefully calibrated hybrid dispatch strategy. Fourth, and most practically, no production-grade open-source antivirus engine has incorporated a native GPU acceleration layer backed by statistically rigorous evaluation, leaving the field without a concrete reference implementation against which these assumptions can be tested.

This thesis addresses all four of these obstacles through a direct engineering intervention in the ClamAV antivirus engine. Beyond raw performance, the architecture is designed with cloud deployment in mind: by encapsulating the GPU scanning logic into stateless, containerized compute modules, the system can be deployed as an event-driven microservice that scales horizontally across available hardware in response to demand, making hardware-accelerated malware scanning viable in serverless cloud environments.

1.3 Research Objectives

This thesis pursues five concrete technical objectives. The first is **automaton mapping and correctness**: designing a translation framework that converts the Aho–Corasick DFA into a representation suitable for GPU execution while preserving functional equivalence with the original engine for all signatures evaluated on the device. The

second is **performance characterisation**: quantifying the throughput improvement delivered by the GPU engine relative to a single-threaded CPU baseline and a commercial multi-threaded reference scanner, across a range of file types and input sizes, using repeated-trial statistics with confidence intervals. The third is **transfer overhead analysis**: identifying empirically the file size threshold above which the GPU execution advantage exceeds the fixed cost of PCIe data transfer, and characterising the relationship between file size distribution and observed speedup across file type categories. The fourth is **logical signature acceleration**: engineering a mechanism for evaluating complex logical signatures — multi-part Boolean conditions that cannot be reduced to simple byte matching — directly on the GPU, without stalling the main parallel scanning pipeline. The fifth is **detection accuracy analysis**: formally measuring true positives, false negatives, false positives, recall, precision, and F_1 score for the GPU engine relative to the CPU ground truth, and identifying and explaining all sources of detection discrepancy.

1.4 Contributions

The main contributions of this thesis are the following:

1. A space-optimized GPU DFA architecture that encodes the dominant ClamAV production signature root — 2,651,902 patterns across 486,426 unique states — into a contiguous 475 MB transition table that resides entirely within the 4 GB VRAM of a commodity AMD Radeon RX 6400 GPU.

2. A flattened DFA memory layout that replaces pointer-linked trie traversal with direct array indexing, eliminating irregular memory access and enabling coalesced global memory reads across GPU wavefronts, together with a root-state local memory cache that serves the dominant state-0 transitions at single-cycle latency.

3. An on-chip logical signature evaluation layer that compiles Boolean sub-signature expressions into stack-based bytecode, tracks sub-signature matches using thread-safe atomic operations, and resolves detection conditions in a dedicated second kernel decoupled from the main scanning pipeline.

4. A statistically rigorous empirical evaluation across ten file type categories and three live malware corpora drawn from the VirusShare repository, reporting mean scan time, standard deviation, and 95% confidence intervals ($n = 10$ runs per configuration), achieving a peak throughput improvement of $\mathbf{15.13}\times$ over the single-threaded CPU baseline on large ELF binary corpora.

5. A full detection accuracy analysis against the CPU ground truth across 3,837 ground-truth positive files, yielding an overall recall of **95.20%**, precision of **100%**, and F_1 score of **0.9754**, with all missed detections attributable to the single-root architectural constraint rather than to errors in the pattern-matching logic.

6. A reduction in host CPU utilization to below **12.5%** during active GPU scans of large-file corpora, freeing processor capacity for concurrent system tasks without performance degradation.

1.5 Novelty

1.5.1 Technical Novelty

The primary technical novelty of this work lies in its scope: rather than constructing an isolated string-matching prototype, this thesis delivers the first GPU acceleration layer integrated directly into a production antivirus codebase, evaluated against the full ClamAV production signature database. This is made possible by a DFA linearization methodology that converts the pointer-linked internal representation of ClamAV's matching engine into a cache-aligned flat matrix amenable to GPU execution. The system further introduces a two-stage verification pipeline in which a high-throughput GPU fast path handles the dominant case of simple byte-pattern matching, while a separate on-chip evaluation unit handles complex logical signatures lazily, without blocking the parallel scanning threads. An empirically determined chunk size schedule — tuned by a factor of four reduction that measurably improved GPU occupancy — and a 256 KB dispatch threshold established through direct measurement further distinguish this implementation from prior prototype work.

1.5.2 Scientific Novelty

On the scientific side, this work provides the first systematic empirical characterisation of GPU-accelerated antivirus scanning across a diverse set of real-world file types and live malware samples, conducted with repeated-trial statistical methodology. By reporting mean, standard deviation, and 95% confidence intervals across ten file type categories and providing complete file size distribution statistics for each corpus partition, the study isolates precisely how file size and content characteristics interact with the GPU dispatch threshold to determine whether offloading is beneficial. Furthermore, by reporting a formal confusion matrix analysis (TP, FN, FP, TN, recall, precision, F_1) for both the GPU engine and a commercial scanner relative to the CPU ground truth, the work establishes the first quantitative detection accuracy benchmark for GPU-accelerated open-source antivirus scanning.

1.6 Industry Relevance

The implications of this work extend across enterprise security and cloud infrastructure. For antivirus vendors, a peak throughput improvement of 15.13× over a single-threaded engine reduces the compute cost of large-scale file auditing and makes real-time inspection viable in contexts where it was previously impractical. The use of an open, hardware-agnostic framework based on OpenCL avoids dependence on proprietary vendor ecosystems and allows the approach to generalise across GPU hardware from different manufacturers, including AMD, Intel, and NVIDIA.

For cloud operators and enterprise security teams, the serverless deployment model enables asynchronous, on-demand scanning that integrates naturally into automated file ingestion pipelines without requiring dedicated persistent infrastructure. Eliminating software licensing costs by building on an open-source foundation reduces the barrier to adoption for organisations operating under budget constraints.

The host CPU utilization finding is practically significant independent of raw speedup. Reducing utilization from ∼100% to below 12.5% during active scans means the host processor remains available for concurrent workloads, allowing the scanning engine to run alongside the network stack, compression pipelines, and logging infrastructure without resource contention. This co-processor model — GPU absorbs the pattern-matching bottleneck, CPU handles everything else — is the natural deploy-

ment architecture for high-throughput network security gateways and cloud storage ingestion systems.

The detection accuracy results place practical limits on the current deployment scope. The 95.20% recall figure is sufficient for a supplementary scanning layer or a first-pass filter, but not for a primary detection engine in a high-stakes environment. The gap is architecturally bounded: loading both the type-specific root and the generic root simultaneously for every file — rather than selecting one or the other, as the current implementation does — would recover the missing coverage, and the path to doing so is clear, requiring either a GPU with larger VRAM or a time-multiplexed multi-root dispatch model.

1.7 Thesis Structure

The remainder of this thesis is organised as follows. Chapter 2 establishes the necessary technical foundations, covering the ClamAV engine architecture, the Aho–Corasick algorithm, the GPU computing model, and the OpenCL programming framework. Chapter 3 surveys related work in GPU-accelerated pattern matching, automaton compression, and hybrid CPU–GPU detection systems, and positions this thesis against five specific gaps in the prior literature. Chapter 4 describes the system design and implementation in detail, including DFA linearization, the root-state local memory cache, the hybrid dispatch layer, chunk size tuning, and the logical signature evaluation unit. Chapter 5 presents the experimental methodology and results, reporting scan time statistics with confidence intervals, file size distributions, detection accuracy analysis, and comparison with the ESET commercial scanner. Chapter 6 interprets the findings, explains the speedup profile across file types, discusses the chunk size tuning effect, situates the results against prior work, and identifies the limitations of the current implementation. Chapter 7 summarises the contributions and outlines directions for future work.

Chapter 2

Background

This chapter establishes the technical foundations underlying the system developed in this thesis. Section 2.1 describes the architecture of ClamAV, its signature database formats, and its default pattern matching pipeline. Section 2.2 formalizes the Aho–Corasick algorithm and analyses its structural limitations under GPU parallelization. Section 2.3 presents the GPU computing model, memory hierarchy, and the optimization principles that motivate the design decisions in subsequent chapters. Section 2.4 introduces the OpenCL programming model, covering its platform hierarchy, execution model, memory regions, and host–device transfer mechanisms as they apply to the system developed in this thesis.

2.1 ClamAV Architecture

2.1.1 Signature Database and CVD Container Format

ClamAV distributes threat indicators through signed, compiled `.cvd` (ClamAV Virus Database) containers. Each container is protected by a cryptographic header that guarantees authenticity and integrity across client endpoints. Upon synchronization and engine initialization, these containers are decompressed and loaded into memory as a set of specialized format structures, each optimized for a distinct class of detection logic.

The Normal Database (`.ndb`) format handles general-purpose signature matching by encoding hex byte sequences augmented with wildcard expressions, case-insensitivity modifiers, and bounded positional offsets. Relational threat rules are expressed through the Logical Database (`.ldb`) format, which chains multiple sub-signatures using Boolean operators to model compound infection conditions. The Hash Database (`.hdb`) stores precomputed MD5, SHA-1, and SHA-256 fingerprints of known malicious files, enabling constant-time exact-match identification before computationally intensive parsing begins. Finally, the Image Fuzzy Hash Database (`.igm`) applies perceptual distance metrics to embedded visual assets in order to detect exploit structures and anomalous graphic payloads that evade byte-level matching.

In the evaluation conducted in this thesis, the full production ClamAV signature database is used without modification, comprising 3,968,449 active signatures compiled into multiple independent Aho–Corasick matcher roots. The GPU engine flattens and uploads exactly one root per scan, selected by the file's detected type: root type 1 (Target:1, Windows PE), encoding 2,651,902 NDB patterns, is loaded for files classified as PE; root type 0 (Target:0, GENERIC, applicable to any file type) is loaded for all other file types evaluated on the GPU. All remaining roots, and — for any given file — whichever of root 0 or the type-specific root was not selected, are evaluated only by the CPU engine. The consequence of this single-root-per-scan architecture for detection coverage is discussed in Chapter 5.

2.1.2 Pattern Matching Engine

The core inspection pipeline relies on the Aho–Corasick algorithm, which compiles a set of target patterns into a single unified finite automaton, enabling simultaneous multi-pattern matching in a single pass over the input. Execution proceeds in two sequential phases.

During the construction phase, the engine builds a forward prefix trie from the compiled pattern set and annotates each node with failure transition links. These links encode, for each state, the longest proper suffix of the current prefix that is also a prefix of some pattern in the dictionary, allowing the automaton to recover from mismatches without backtracking in the input stream. During the scanning phase, bytes from the target file buffer are consumed sequentially to drive state transitions through the automaton. When a transition reaches an accepting state, a match event

is emitted and recorded, completing detection in $O(n)$ time with respect to input length regardless of the number of patterns.

2.1.3 Logical Signatures

Logical signatures represent compound threat indicators that express detection conditions beyond static byte alignment. A representative definition of the form

```
Win.Trojan.Generic-6628741-0;Engine:51-255;Target:1;
    0&1&2&3&4;subsig0;subsig1;subsig2;subsig3;subsig4
```

requires that five distinct sub-signatures all match within a single target file before a detection event is raised, and constrains evaluation to Target type 1 (Windows PE). This structure allows analysts to encode behavioural relationships, structural co-occurrence constraints, and file-type restrictions that reduce false positive rates compared to single-pattern rules.

The evaluation model decouples sub-signature matching from Boolean expression resolution. As the scanning unit processes the input stream, it maintains a presence vector tracking which sub-signatures have been satisfied. The final Boolean expression is evaluated only after the complete file has been consumed, ensuring that relational logic evaluation does not interrupt the byte-level matching pipeline. In the GPU implementation, this two-phase model is preserved by separating the main DFA scan kernel from a second logical evaluation kernel, as described in Chapter 4.

2.1.4 Multiple Matcher Roots and File-Type Targeting

ClamAV does not compile all signatures into a single automaton. Instead, it organises signatures into multiple independent Aho–Corasick roots, one per target file type, each constructed and stored separately. A signature's `Target` field determines which root it belongs to: Target 0 covers any file type, Target 1 covers Windows PE executables, Target 6 covers ELF binaries, and so on.

This per-root organisation has a direct consequence for the GPU implementation: each root would need to be independently flattened and uploaded to device memory to achieve full simultaneous coverage. At the scale of the ClamAV database, root type 1 alone occupies 475 MB, leaving insufficient VRAM on the evaluation device to hold every root at once. The current implementation therefore loads exactly one

flattened root per scan: root type 1 for files classified as Windows PE, and root type 0 (the generic root, whose `Target` field of `CL_TYPE_ANY` makes it applicable to every file type) for all other files evaluated on the GPU. A consequence of this design is that, unlike the CPU engine — which always evaluates root 0 in addition to whichever type-specific root applies to a given file — the GPU evaluates only one of the two for any single file: PE files are scanned against root 1 but never root 0 on the GPU, while every other file type is scanned against root 0 but never its own type-specific root (e.g. root 6 for ELF) on the GPU. CPU fallback handles signatures in whichever root was not loaded for a given file. The practical effect on detection recall is quantified in Chapter 5.

2.2 Aho–Corasick Algorithm

2.2.1 Formal Definition

Given a pattern dictionary $P = \{p_1, p_2, \ldots, p_k\}$, the Aho–Corasick automaton is the 5-tuple $(Q, \Sigma, \delta, q_0, F)$. Q is the finite set of states, corresponding to the nodes of the prefix trie constructed from P. $\Sigma = \{0\text{x}00, \ldots, 0\text{x}\texttt{FF}\}$ is the input alphabet over the 8-bit byte space. $\delta : Q \times \Sigma \to Q$ is the total state transition function, defined by combining the goto function of the trie with failure links to ensure δ is defined for every (q, c) pair. $q_0 \in Q$ is the initial state, corresponding to the trie root. $F \subseteq Q$ is the set of accepting states, each annotated with the set of patterns whose occurrence it signals.

The construction of δ proceeds in two passes. The first pass builds the trie and assigns failure links via breadth-first traversal. The second pass resolves the full transition function so that $\delta(q, c)$ is defined without requiring recursive failure-link traversal at runtime, yielding a complete deterministic finite automaton (DFA).

2.2.2 Complexity Analysis

Automaton construction requires $O\left(\sum_{i=1}^{k} |p_i|\right)$ time and space, scaling linearly with the total length of all patterns. Once constructed, the scanning phase processes an input of length n in $O(n)$ time, since each byte causes exactly one state transition and the DFA formulation eliminates failure-link traversal at runtime. This independence

from dictionary size is the principal theoretical advantage of the algorithm: throughput remains asymptotically constant whether matching against tens or millions of concurrent patterns.

At the scale used in this thesis — 2,651,902 patterns across 486,426 states in root type 1, the larger of the two roots loaded by the GPU engine — the DFA transition table occupies 475 MB.

2.2.3 Limitations for GPU Parallelization

Despite its optimal sequential complexity, the conventional Aho–Corasick implementation is structurally ill-suited for execution on graphics processors. In a pointer-linked trie representation, evaluating $\delta(q, c)$ requires fetching a node structure, testing whether a direct child exists for the input character c, and — on a mismatch — following a chain of failure links until a valid transition is found. Each of these steps dereferences a pointer to a non-contiguous heap address, producing highly irregular memory access patterns.

This irregularity conflicts directly with the requirements of the Single Instruction, Multiple Threads (SIMT) execution model described in Section 2.3.1. Threads within a wavefront will generally be processing different bytes and residing at different automaton states, causing them to follow divergent pointer chains through disjoint memory regions. The resulting warp divergence serializes what should be parallel execution, while the scattered memory references defeat cache prefetching and cause repeated high-latency VRAM accesses. Both effects substantially reduce effective throughput relative to the theoretical peak of the device.

The solution adopted in this thesis is DFA flattening: converting the pointer-linked trie into a contiguous two-dimensional array where `next_state` = `dfa[state` $\times$ $256 +$ `input_byte]`. This layout enables coalesced global memory reads and eliminates pointer indirection entirely, at the cost of a fixed 475 MB memory footprint for the dominant root.

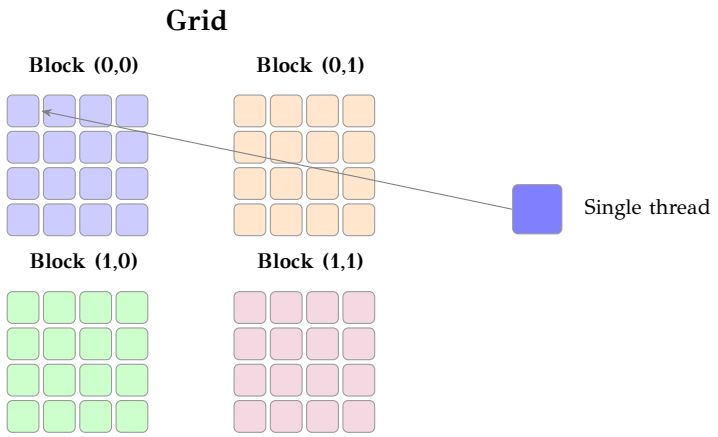

Figure 2.1: GPU SIMT execution hierarchy. Each cell is one thread. Threads in a row form a warp (32 threads executing in lockstep). Blocks share L1 cache; all blocks share global VRAM.

2.3 GPU Computing Architecture

2.3.1 SIMT Execution Model

Modern GPU architectures achieve high computational throughput by organizing processing resources into a hierarchical, data-parallel execution model known as Single Instruction, Multiple Threads (SIMT). At the lowest level, individual threads execute scalar operations independently, each holding its own register state. Threads are grouped into *warps* (NVIDIA terminology) or *wavefronts* (AMD terminology) that execute in lockstep: all threads in a group issue the same instruction each cycle, applied to their respective data. On the AMD Radeon RX 6400 used in this thesis, wavefronts are 64 work-items wide. Multiple wavefronts are assigned to a *work-group* (OpenCL) or *thread block* (CUDA), which has access to a shared, programmer-managed on-chip local memory space used for intra-group communication and data reuse. Work-groups are organized into an NDRange that distributes work across the physical compute units of the device.

The efficiency of this model depends critically on thread uniformity. When threads within a wavefront follow different control flow paths — a condition known as warp divergence — the hardware executes each divergent path serially while masking the inactive threads, reducing effective parallelism in proportion to the number of distinct paths taken. This is the primary mechanism by which PE executables underperform ELF binaries in the GPU scanner: packed PE content triggers the verification pipeline

branch far more frequently, causing divergence within wavefronts that ELF scanning largely avoids.

2.3.2 Memory Hierarchy

The GPU memory subsystem is organized into a hierarchy of storage tiers that trade capacity for access latency. The register file provides single-cycle access but is limited to a few hundred kilobytes per compute unit. The L1 and L2 cache levels absorb irregular access patterns with latencies on the order of 5 to 100 cycles. Off-chip VRAM, implemented as high-bandwidth GDDR6 memory on the RX 6400, provides 4 GB capacity but incurs access latencies of 300 to 500 cycles. The practical consequence of this hierarchy is that sustained throughput depends almost entirely on data locality: computations whose working sets fit within L1 or L2 cache can approach theoretical peak throughput, while workloads that repeatedly access VRAM are bottlenecked by memory latency regardless of arithmetic intensity.

In the context of this thesis, the 475 MB DFA transition table far exceeds L2 cache capacity, making most transitions VRAM-bound. The root-state local memory cache (Section 4.5.1) partially compensates for this by serving the dominant state-0 transitions from on-chip local memory, but non-zero state transitions remain subject to VRAM latency.

2.3.3 Optimization Principles

Four principles govern the translation of an algorithm into an efficient GPU implementation. Each is directly relevant to the design decisions described in Chapter 4.

Coalesced memory access. When adjacent threads in a wavefront read or write contiguous memory addresses, the hardware issues a single wide memory transaction covering all requests. The flattened DFA layout enables coalesced reads: all threads in a work-group that share the same current state read contiguous entries in the transition table.

Minimization of warp divergence. Conditional branches whose outcome varies across threads in a wavefront serialize execution. The verification pipeline in the GPU scanner is the primary source of divergence: threads that encounter DFA accepting states branch into the verification path while neighbours continue scanning. Reducing the rate of false accepting-state hits — which packed PE content produces in

abundance — is therefore the key to improving PE speedup.

Occupancy management. Occupancy, defined as the ratio of active wavefronts to the maximum supported by the hardware, determines the degree to which memory latency can be hidden by computation from other wavefronts. The chunk size reduction described in Chapter 4 directly increases occupancy by launching more work-items per file.

Data locality. Structuring data layouts so that iterative access patterns target on-chip cache lines reduces dependence on high-latency VRAM transactions. The root-state local memory cache exploits this principle by placing the 256 most-accessed DFA transitions in on-chip local memory, achieving single-cycle access for the state-0 fast path.

2.3.4 Gap in Existing Research

While prior work has demonstrated that individual components of deterministic automata can be accelerated on GPU hardware, a significant gap remains between academic prototype systems and full enterprise deployments. Existing approaches commonly evaluate against simplified, artificially constrained signature sets that omit the compound Boolean logic and wildcard operators present in production databases. No prior system has reported repeated-trial statistics with confidence intervals, and none has evaluated against a signature database at the scale of ClamAV's 3,968,449-signature corpus. These limitations motivate the design decisions and evaluation methodology described in subsequent chapters.

2.4 OpenCL Programming Model

2.4.1 Overview and Motivation

OpenCL (Open Computing Language) is an open, royalty-free standard for parallel programming across heterogeneous hardware platforms, maintained by the Khronos Group. Unlike vendor-specific frameworks such as NVIDIA CUDA, which targets only NVIDIA GPUs, OpenCL programs execute on any conforming device, including GPUs from AMD, Intel, and NVIDIA, as well as multi-core CPUs and FPGAs. This hardware agnosticism is the primary motivation for its use in this thesis: the acceleration

layer must remain portable across the diverse GPU hardware configurations found in enterprise and cloud environments, where no single vendor can be assumed. The evaluation was conducted on an AMD Radeon RX 6400, a device that CUDA cannot target, making OpenCL the only viable framework for this hardware.

2.4.2 Platform and Device Model

OpenCL organizes hardware into a two-level hierarchy. At the top level, a *platform* represents a vendor's OpenCL runtime installation, such as the AMD ROCm platform or the Intel OpenCL runtime. Each platform exposes one or more *devices*, which correspond to individual physical processing units. A device is further subdivided into *compute units*, each of which maps to a group of parallel execution lanes on the underlying hardware. On AMD RDNA2 hardware (as used in the RX 6400), a compute unit contains 64 shader processors organized as a single 64-wide wavefront execution unit.

At runtime, an application selects a platform and device, then creates a *context* that manages memory objects and synchronization state across that device. All data transfers and kernel executions are submitted through a *command queue* bound to the context, which the runtime schedules onto the device asynchronously.

2.4.3 Execution Model

Parallel work in OpenCL is expressed through *kernels*: functions written in OpenCL C, a dialect of ISO C99 extended with parallel execution qualifiers and built-in vector types. When a kernel is enqueued for execution, the programmer specifies an *NDRange* (N-Dimensional Range) that defines the total number of work-items to launch. Work-items are grouped into *work-groups*, which execute on a single compute unit and share access to a fast, programmer-managed local memory region. Within a work-group, a fixed-size subset of work-items executes in lockstep as a *wavefront* (typically 64 work-items wide on RDNA2 hardware).

In the GPU scanner, each work-item processes one chunk of the input file. The NDRange size is therefore $\lceil L/stride \rceil$ for a file of length L, where stride is the chunk size minus the pattern overlap. The chunk size schedule described in Chapter 4 directly controls the NDRange size and therefore the number of active wavefronts, which is the primary lever for GPU occupancy.

2.4.4 Memory Model

OpenCL defines four distinct memory regions that map onto the GPU memory hierarchy described in Section 2.3.2.

Global memory is the largest region, corresponding to off-chip VRAM. It is accessible by all work-items across all work-groups but carries the highest access latency (300–500 cycles on RDNA2). The DFA transition table, pattern metadata, and file input buffer all reside in global memory.

Local memory is a fast, on-chip region shared among all work-items within a single work-group. It is explicitly managed by the programmer and used to stage data that will be accessed repeatedly. In the GPU scanner, the first 256 entries of the DFA transition table (the root-state transitions) are staged in local memory at kernel entry, as described in Section 4.5.1.

Private memory is per-work-item storage backed by the physical register file. It holds the current DFA state, loop indices, and match coordinates, and provides single-cycle access latency.

Constant memory is a read-only region cached close to the execution units. It is suitable for data that is uniform across all work-items, such as the `boundary_table[256]` used in the boundary validation routines.

In the context of this thesis, the DFA state transition table is allocated as a global memory buffer. Because its access pattern is data-dependent — each work-item reads from an offset determined by its current state and input byte, which varies unpredictably for non-zero states — the table does not benefit from hardware prefetching for the general case. The root-state local memory cache recovers prefetch-equivalent performance for the state-0 case, which dominates for ELF and APK corpora but not for packed PE content.

Chapter 3

Related Work

Signature-based malware detection has attracted sustained research interest in GPU acceleration, automaton compression, and hybrid execution models. Despite this body of work, no prior system has successfully integrated GPU-accelerated pattern matching into a production antivirus codebase, evaluated against a full-scale signature database with repeated trials and confidence intervals, or validated correctness across a diverse set of real-world malware corpora spanning ten file type categories. This chapter traces the development of the field, identifies the specific gaps that each line of research left open, and explains how the contributions of this thesis address them.

3.1 Foundations of Multi-Pattern String Matching

The Aho–Corasick algorithm [1] is the algorithmic foundation of virtually every production signature-based antivirus engine. By compiling an entire pattern dictionary into a single finite automaton and processing input in one left-to-right pass, it achieves $O(n)$ scanning complexity independent of the number of patterns. Al-Dabbagh et al. [2] confirm that this algorithm remains dominant in modern antivirus engines despite the rise of machine-learning-based detection, precisely because its deterministic complexity guarantee is essential for high-volume scanning environments.

The theoretical attractiveness of Aho–Corasick conceals a practical problem that is central to this thesis. Kumar et al. [3] showed that constructing a complete DFA over large rule sets produces state spaces that exceed available memory, even on host

systems. On a GPU, where VRAM is an order of magnitude smaller than host RAM and memory latency penalties are far more severe, this problem becomes the primary obstacle to deployment. No prior work solved it at production database scale; this thesis does, by compressing the larger of ClamAV's two GPU-loaded matchers — type 1, with 2,651,902 patterns across 486,426 states — into a 475 MB flat transition table that fits entirely within the 4 GB VRAM of the evaluation device, alongside a smaller generic-root (type 0) table loaded for non-PE scans.

3.2 GPU-Accelerated Pattern Matching Systems

The case for GPU-accelerated pattern matching was established by Owens et al. [4] and Nickolls and Dally [5], who identified data-parallel string matching as a natural fit for the SIMT execution model. The first concrete antivirus application was Vasiliadis, Ioannidis et al.'s Gnort [6], which offloaded Snort signature matching to a GPU, followed by their Gravity engine [7], which extended the approach to antivirus scanning and reported end-to-end throughput approximately 100× that of CPU-only ClamAV, with a separate pre-cached-data micro-benchmark reaching roughly double that figure. Zha and Sahni [8] and Lin et al. [9] provided complementary analyses of partitioning strategies and thread block organization for GPU string matching.

A sustained research program on GPU-parallel Aho–Corasick was carried out by Tran et al. [10, 11, 12, 13], who progressively identified and addressed the primary bottleneck: irregular memory traversal during failure-link transitions. Yoon et al. [14] proposed access reordering to reduce global memory latency, and Scarpazza et al. [15] demonstrated similar lessons on the Cell/B.E. processor.

The critical limitation shared by all of these systems is that they are research prototypes evaluated against small, hand-crafted signature sets. None operated against a production database at the scale of ClamAV's 3,968,449-signature corpus, none were integrated into a working antivirus engine, and none reported repeated-trial statistics with confidence intervals. The performance figures they report cannot be directly compared to the baselines that matter in practice: a fully loaded production engine scanning real malware files over repeated runs. This thesis fills that gap, reporting mean scan times, standard deviations, and 95% confidence intervals across $n = 10$ runs per configuration throughout the evaluation in Chapter 5.

3.3 Automaton State Space Compression

The state space problem has been approached from several directions. Becchi and Crowley [16] proposed hybrid finite automaton representations to limit the memory explosion when converting NFA to DFA for deep packet inspection. Bellekens et al. [17] developed memory-compression schemes targeted specifically at GPU-accelerated intrusion detection, reducing transition table footprints. Liu et al. [18] explored asynchronous automata processing on GPUs, demonstrating that decoupling data ingestion from state traversal reduces stall cycles.

These contributions address memory pressure at the level of individual components. None of them tackled the full ClamAV database, which presents a qualitatively different problem: the type 1 matcher alone has 486,426 states and the complete database spans multiple independent roots of varying sizes, each requiring separate construction and upload. The per-root architecture developed in this thesis — which flattens each root independently and uploads exactly one per scan, selected by file type, to stay within GPU memory limits — is a direct engineering response to a constraint that prior compression work never encountered at this scale. Crucially, this design also exposes the primary limitation of the current implementation: for any given file, only one of the two roots the CPU engine would normally apply is loaded onto the GPU, which accounts for the missed-detection rate observed in Chapter 5. Extending coverage to additional simultaneously-loaded roots is the most impactful direction for future work, as discussed in Chapter 6.

3.4 Hybrid CPU–GPU Execution Architectures

Several works recognized that GPU implementations cannot handle all pattern types uniformly. Velea et al. [19] proposed routing signatures between CPU and GPU based on structural complexity. Pungila and Negru [20] focused on hybrid automaton construction. Kouzinopoulos et al. [21] combined Aho–Corasick and Wu–Manber in a CUDA and MPI hybrid evaluated on biological sequence data.

These hybrid designs share a common limitation: the dispatch criterion is based on pattern class alone, determined at database load time. They do not account for the fixed per-file cost of PCIe data transfer, which means they provide no principled answer to the question of when GPU offloading actually pays off for a given input.

This thesis addresses this directly by establishing a 256 KB file-size threshold through empirical profiling. The threshold's practical effect is demonstrated by the file size distribution tables in Chapter 5: file types whose mean size falls below 256 KB — GIF (mean 54–59 KB), JPEG (mean 65–70 KB), JavaScript (mean 36–39 KB), XML (mean 43–46 KB) — consistently show GPU slowdowns, while types with mean sizes well above the threshold — ELF (mean 1.49–2.41 MB), PE (mean 1.26–1.63 MB), APK (mean 3.03–9.30 MB) — show consistent speedups.

3.5 Hardware-Oriented Detection Approaches

For completeness, dedicated hardware approaches provide the upper bound on matching throughput. Sidhu and Prasanna [22] implemented regular expression matching directly in FPGA logic, achieving line-rate throughput by eliminating all software overhead. Fechner [23] applied the same principle to Aho–Corasick for malware detection at the circuit level. Aldwairi et al. [24] characterized the trade-offs between reconfigurability and throughput for Wu–Manber hardware.

These systems are not practically deployable for antivirus use because they cannot be updated when the signature database changes — a process that happens daily in production environments. The GPU-based approach in this thesis retains full reprogrammability: updating the signature database requires only rebuilding the flat DFA arrays at load time, with no hardware changes.

3.6 OpenCL as an Acceleration Framework

Stone et al. [25] characterized OpenCL as a portable parallel programming standard capable of performance competitive with vendor-specific frameworks such as CUDA across a range of hardware. The GPU-based systems described in prior sections were almost exclusively built on CUDA, tying them to NVIDIA hardware. OpenCL was chosen for this thesis precisely to avoid that constraint: the acceleration layer must operate in enterprise and cloud environments where GPU hardware vendor cannot be assumed, and where a CUDA dependency would rule out AMD and Intel devices entirely. The evaluation was conducted on an AMD Radeon RX 6400, a device that

CUDA cannot target, confirming the practical necessity of this choice.

3.7 Positioning of This Work

The prior literature establishes that GPU acceleration of Aho–Corasick pattern matching is theoretically sound and that individual components — DFA flattening, memory layout optimization, hybrid dispatch — can each yield measurable throughput improvements. What the literature does not provide is a system that combines all of these components under real operational conditions with statistically rigorous evaluation. Five specific gaps distinguish this thesis from prior work.

The first is **production codebase integration**. Every prior GPU-accelerated antivirus or intrusion detection system is a standalone prototype. This thesis modifies ClamAV 1.0.0 directly, which means all results are obtained under the same conditions as a deployed engine: full signature database loading across multiple matcher roots, file-format-specific scan dispatchers, ClamAV's own logical signature evaluation semantics, and production memory management constraints.

The second is **full-scale database evaluation**. Prior systems evaluate against tens of thousands of patterns at most. This thesis operates against 3,968,449 signatures organized across multiple independent DFA roots, with the dominant root alone containing 486,426 states. The per-root upload architecture and the 475 MB memory footprint are direct consequences of working at this scale, and they have no analogue in prior work.

The third is **empirical threshold determination**. Prior hybrid systems partition work by pattern type but provide no empirical basis for deciding when GPU offloading is beneficial for a given file. This thesis identifies 256 KB as the crossover point through direct measurement and implements a size-based dispatch that routes files below this threshold to the CPU regardless of their content. The file size distribution analysis in Chapter 5 provides the first systematic demonstration of how this threshold interacts with real-world file type distributions across diverse malware corpora.

The fourth is **statistical rigour**. No prior work in GPU-accelerated antivirus scanning reports repeated-trial statistics; all performance figures in prior systems are single-run measurements. This thesis conducts $n = 10$ independent runs per configuration and reports mean scan time, standard deviation, and 95% confidence intervals

throughout, making the results directly comparable across hardware generations and reproducible by independent researchers.

The fifth is **diverse real-world evaluation with detection accuracy analysis**. Prior systems evaluate on narrow, often synthetic corpora, and none report formal detection accuracy metrics. This thesis validates throughput across ten distinct file categories drawn from live malware samples in the VirusShare repository [26] — ELF binaries, Windows PE executables, APK packages, HTML documents, JavaScript, XML, JPEG, PNG, GIF, and plain text — and provides a full confusion matrix analysis (TP, FN, FP, TN, recall, precision, F_1) for both the GPU engine and the ESET commercial scanner relative to the CPU ground truth, yielding the first systematic characterisation of how file-type characteristics affect GPU scanning accuracy and throughput simultaneously.

Chapter 4

Implementation

This chapter describes the implementation of the GPU-accelerated pattern matching pipeline integrated into ClamAV. The implementation modifies ClamAV's Aho–Corasick matcher construction, database loading path, runtime scan dispatcher, and pattern verification engine in order to offload large-file signature scanning to the GPU while preserving compatibility with the original CPU engine. It proceeds through seven stages: DFA construction and flattening, pattern metadata encoding, GPU runtime memory allocation, hybrid GPU scan execution, GPU-side pattern verification, logical signature evaluation, and CPU fallback and compatibility handling.

4.1 Step 1: DFA Construction and GPU Preparation

During database loading, ClamAV builds one Aho–Corasick matcher per target type. GPU preparation was integrated directly into `cli_ac_buildtrie()` after the normal CPU trie construction phase.

4.1.1 State Space and Memory Footprint

ClamAV organizes its Aho–Corasick automata into multiple independent matcher roots, one per target file type, compiled separately during database loading. Due to GPU memory constraints, only one flattened root is resident on the device at a time, selected per scan according to the file's detected type: root type 1 for files classified

as Windows PE, and root type 0 — ClamAV's generic root, whose `Target` field covers any file type — for all other files evaluated on the GPU. This architectural decision is the primary source of the missed-detection rate reported in Chapter 5: for any given file, signatures residing in the root that was *not* selected for that scan (including root 0 itself, for PE files) are evaluated only by the CPU engine.

Root type 0, the generic root loaded for all non-PE files evaluated on the GPU, is substantially smaller: it encodes 31,582 patterns across 11,623 unique trie states. Following the same flattening methodology as root 1, its transition table has $11,623 \times 256 = 2,975,488$ entries which, stored as 32-bit integers, occupy approximately 11.35 MB. Including its output index and output count auxiliary arrays, the total root 0 footprint on device memory is approximately 11.4 MB — roughly 42 times smaller than root 1's 475 MB transition table. This leaves substantial VRAM headroom when root 0 is the active root for a scan, though the two roots are never resident simultaneously in the current implementation (Section 2.1.4).

The dominant root, type 1, handles the majority of NDB signatures and encodes 2,651,902 patterns expanding to 486,426 unique trie states after prefix sharing. Its transition table has $486,426 \times 256 = 124,524,736$ entries which, stored as 32-bit integers, occupy approximately 498 MB uncompressed. After state remapping to eliminate unreachable states, the working footprint is reduced to 475 MB, fitting within the 4 GB VRAM of the AMD Radeon RX 6400. The full database comprises 3,968,449 signatures distributed across all roots combined.

The auxiliary output arrays for the type 1 root occupy an additional $\sim$8 MB: the output index array ($486,426 \times 4$ bytes $\approx$ 1.95 MB), the output count array ($486,426 \times 2$ bytes $\approx$ 1 MB), and the flat output pattern list (1,002,253 entries $\times$ 4 bytes $\approx$ 3.8 MB). The total device allocation for the type 1 matcher therefore sits at approximately 483 MB, leaving over 3.5 GB of VRAM available for file buffers and tracker pools. By comparison, root 0 (Section 4.1.1), the smaller root loaded for non-PE scans, occupies only approximately 11.4 MB — a difference that reflects the relative sizes of ClamAV's PE-targeted and generic signature sets rather than any difference in flattening methodology.

4.1.2 GPU Initialization Sequence

The GPU initialization stage performs four operations: assigning contiguous GPU pattern identifiers, building a GPU lookup table, flattening the DFA transition graph, and uploading the resulting GPU structures to device memory.

Algorithm 4.1 summarises the initialization sequence.

Algorithm 4.1 GPU matcher initialization integrated into `cli_ac_buildtrie()`

Require: Completed CPU trie root after `link_lists()` and `ac_maketrans()`

Ensure: Flattened DFA and pattern metadata uploaded to GPU device memory

1: Build CPU trie: link_listsroot; ac_maketransroot
2: $gpu\_id \leftarrow 0$
3: **for** each pattern p in `ac_pattable[]` **do**
4: $p.gpu\_id \leftarrow gpu\_id$; $gpu\_id \leftarrow gpu\_id + 1$
5: **end for**
6: Allocate `gpu_patt_lookup[gpu_patt_count]`
7: **for** each pattern p with assigned `gpu_id` **do**
8: `gpu_patt_lookup`[$p.gpu\_id$] $\leftarrow p$
9: **end for**
10: `gpu_flat_dfa` $\leftarrow$ gpu_build_flattened_dfaroot
11: gpu_rt_upload_flattened_dfa`gpu_flat_dfa`
12: gpu_upload_pattern_metadata`gpu_patt_lookup`
13: gpu_upload_logical_signatures
14: root.`gpu_enabled` $\leftarrow 1$; root.`gpu_dfa_ready` $\leftarrow 1$

The implementation begins after the standard CPU matcher construction:

```
link_lists(root);
ac_maketrans(root);
```

After the CPU transition graph is finalized, each pattern is assigned a contiguous GPU identifier:

```
for (uint32_t i = 0; i < root->ac_patterns; i++) {
    if (root->ac_pattable[i]) {
        root->ac_pattable[i]->gpu_id = new_gpu_id++;
    }
```

```
}
```

The original ClamAV matcher stores patterns sparsely inside `ac_pattable[]`. GPU execution cannot efficiently traverse sparse pointer structures, therefore the implementation constructs a compact lookup table:

```
root->gpu_patt_lookup =
    calloc(root->gpu_patt_count,
           sizeof(struct cli_ac_patt *));
```

Each GPU pattern ID directly indexes this lookup table:

```
root->gpu_patt_lookup[patt->gpu_id] = patt;
```

This removes pointer traversal during GPU match resolution and allows pattern metadata to be transferred as contiguous arrays.

The DFA flattening stage converts the pointer-based trie into a dense transition table:

```
root->gpu_flat_dfa = gpu_build_flattened_dfa(root);
```

The resulting structure stores all state transitions in contiguous memory:

```
next_state = dfa[state * 256 + input_byte]
```
(4.1)

instead of recursively traversing heap-allocated trie nodes.

Once flattening succeeds, the matcher is marked GPU-ready:

```
root->gpu_enabled = 1;
root->gpu_dfa_ready = 1;
```

The flattened DFA stores one transition entry for every $(state, byte)$ pair. Missing transitions are resolved during construction using the failure-link transitions computed by `ac_maketrans()`, eliminating runtime failure traversal during scanning.

The final GPU DFA representation contains a flattened transition table, an output pattern index table, an output pattern count table, and a contiguous output pattern list.

This converts the original pointer-heavy matcher into a contiguous GPU-compatible representation.

4.2 Step 2: Pattern Metadata Encoding

The DFA transition table identifies candidate terminal states but does not contain enough information to validate complete ClamAV signatures. Additional pattern metadata is therefore transferred separately to the GPU.

Each signature is encoded as a fixed-size `gpu_pattern_t` structure:

```
typedef struct {
    uint length;
    uint prefix_length;
    uint parts;
    uint type;
    uint sigid;
    uint partno;

    uint offset_min;
    uint offset_max;

    uint ch0;
    uint ch1;

    uint ch_mindist0;
    uint ch_maxdist0;

    uint ch_mindist1;
    uint ch_maxdist1;

    uint pattern_offset;
    uint prefix_offset;

    uint virname_offset;
    uint virname_len;

    uint boundary;
```

```
    uint has_regex;
    uint is_bytecode;
} gpu_pattern_t;
```

The structure stores all information required for GPU-side verification without host interaction. Fixed-width fields are used intentionally to avoid variable-length parsing inside GPU kernels.

Pattern byte sequences are stored separately inside a packed 16-bit array. Each entry encodes both the byte value and matching mode: the upper 8 bits carry the matching mode and the lower 8 bits carry the byte value.

The implementation supports the same byte semantics as ClamAV's CPU matcher:

```
GPU_MATCH_CHAR
GPU_MATCH_IGNORE
GPU_MATCH_NOCASE
GPU_MATCH_NIBBLE_HIGH
GPU_MATCH_NIBBLE_LOW
```

Virus names are stored in a contiguous string pool. Pattern descriptors store offsets into this pool rather than pointers:

```
virname_offset
virname_len
```

This avoids GPU-side dynamic allocation and allows the host to resolve virus names directly after a successful detection.

The `has_regex` path is present in the metadata structure but the PCRE fallback branch in the kernel is currently disabled; PCRE patterns with no container restriction are instead handled by the generic container-mismatch check described in Section 4.7. Signatures containing PCRE that specify a container restriction matching the scanned file's type are not caught by this check and will produce a missed detection rather than a fallback, which is one contributor to the detection gaps reported in Chapter 5.

4.3 Step 3: GPU Runtime Allocation

GPU buffers are allocated once during engine initialization and reused across scans. The upload sequence is integrated into `readdb.c` immediately after matcher construc-

tion:

```
gpu_rt_upload_flattened_dfa(...)
gpu_upload_pattern_metadata(...)
gpu_upload_logical_signatures(...)
```

The implementation uploads three major data regions:

Table 4.1: GPU runtime buffers

Buffer	**Contents**
Flattened DFA	State transition table
Pattern metadata	`gpu_pattern_t` descriptors
Logical signature tables	Expression bytecode and metadata

Logical signatures are optional. Upload failure does not abort GPU initialization:

```
cli_warnmsg("GPU: Logical signatures upload failed")
```

This allows the GPU scanner to continue operating even when logical signature support is unavailable. The matcher is enabled only after all mandatory uploads succeed:

```
root->gpu_enabled = 1;
```

Otherwise execution automatically falls back to the CPU engine.

All GPU buffers are persistent and reused across scans. The runtime therefore avoids repeated allocation and deallocation overhead during continuous scanning workloads.

4.4 Step 4: Hybrid GPU Scan Execution

GPU scanning is integrated directly into ClamAV's normal matcher pipeline inside `matcher.c`. Before launching the GPU scan kernel, the runtime verifies that GPU acceleration is enabled, that the DFA upload succeeded, that GPU runtime state exists, and that the input file is large enough to amortize GPU overhead.

Algorithm 4.2 describes the hybrid dispatch logic.

Algorithm 4.2 Hybrid GPU/CPU dispatch in `matcher.c`

Require: File map `fmap`, engine context `ctx`

Ensure: File scanned by GPU engine or CPU fallback; virus name reported if found

if `gpu_enabled` **and** `dfa_uploaded` **and** `fmap.len` $\geq 256 \times 1024$ **then**
  buf $\leftarrow$ fmap_need_off_once`fmap`, 0, `fmap.len`
  result $\leftarrow$ gpu_scan`ctx`, *buf*, `fmap.len`
  if *result* = `GPU_RESULT_VIRUS` **then**
    cli_append_virus`ctx`, `gpu_virname`
    `ctx.gpu_ac_done` $\leftarrow$ *true*
    return INFECTED
  else if *result* = `GPU_RESULT_CLEAN` **then**
    `ctx.gpu_ac_done` $\leftarrow$ *true*
    return CLEAN
  else
    {GPU_RESULT_BREAK: unsupported condition}
    goto `cpu_scan`
  end if
end if
cpu_scan: execute standard ClamAV CPU Aho–Corasick scan

The dispatch condition is:

```
if (generic_ac_root &&
    generic_ac_root->gpu_enabled &&
    ctx->engine->gpu_rt &&
    ctx->engine->gpu_rt->dfa_uploaded &&
    ctx->fmap &&
    ctx->fmap->len >= 256 * 1024)
```

Files smaller than 256 KB are processed entirely by the CPU engine because GPU launch overhead exceeds the expected acceleration benefit. The 256 KB value is an empirically determined crossover point: below it, the fixed costs of PCIe transfer, kernel launch latency, and buffer initialization exceed the time savings from parallel DFA traversal. The effect of this threshold is demonstrated concretely by the file size

distribution tables in Chapter 5: every file type whose mean size falls below 256 KB shows a GPU slowdown.

If GPU execution is selected, the entire file is mapped into memory:

```
const unsigned char *full_buf =
    fmap_need_off_once(ctx->fmap, 0, ctx->fmap->len);
```

The scan is then executed through:

```
gpu_scan(...)
```

The GPU scanner returns one of three execution states:

Table 4.2: GPU scan return states

Return code	**Meaning**
`GPU_RESULT_VIRUS`	Signature matched
`GPU_RESULT_CLEAN`	No matches detected
`GPU_RESULT_BREAK`	Unsupported condition encountered

If a signature is detected, the virus name returned by the GPU runtime is inserted into ClamAV's normal reporting pipeline:

```
ret = cli_append_virus(ctx, gpu_virname);
```

If the GPU encounters a condition that cannot be safely evaluated on-device, execution immediately falls back to the CPU engine:

```
goto cpu_scan;
```

This guarantees that no supported signature type is silently skipped. The GPU runtime also tracks whether GPU scanning completed successfully:

```
ctx->gpu_ac_done = true;
```

This prevents duplicate CPU-side rescanning after successful GPU execution.

4.5 Step 5: GPU-side Pattern Verification

The DFA stage identifies candidate terminal states but does not guarantee an exact signature match. Final verification is therefore performed by GPU-side validation routines implemented in OpenCL.

Algorithm 4.3 describes the per-work-item DFA scanning kernel, which runs in parallel across all chunks of the input file.

Algorithm 4.3 GPU OpenCL scanning kernel (one work-item per chunk)

Require: File buffer `buf`, chunk offset `off`, chunk length `len`, flattened DFA `dfa`, pattern metadata `patterns`

Ensure: Matching virus names written to output tracker

1: Load root-state transitions into `__local` cache `state0_cache[256]`
2: barrier`CLK_LOCAL_MEM_FENCE`
3: $s \leftarrow 0$ {start in root state}
4: **for** $i \leftarrow$ *off* **to** *off* $+$ *len* $- 1$ **do**
5: $c \leftarrow$ *buf*$[i]$
6: **if** $s = 0$ **then**
7: $s \leftarrow$ *state0_cache*$[c]$ {fast path: local memory}
8: **else**
9: $s \leftarrow$ *dfa*$[s \times 256 + c]$ {slow path: global VRAM}
10: **end if**
11: **if** output_count$s > 0$ **then**
12: **for** each pattern p at output state s **do**
13: **if** gpu_findmatchbuf, i, p **then**
14: record_matchp.virname_offset
15: **end if**
16: **end for**
17: **end if**
18: **end for**

The primary verifier is `verify_pattern()`, which validates both backward prefix constraints and forward pattern body matching. Byte comparison is delegated to `match_pattern_byte()`, which implements ClamAV's matching semantics directly on the GPU:

```
GPU_MATCH_CHAR       -- exact byte match
GPU_MATCH_IGNORE     -- wildcard, always matches
GPU_MATCH_NOCASE     -- case-insensitive byte match
GPU_MATCH_NIBBLE_HIGH -- high nibble match
GPU_MATCH_NIBBLE_LOW  -- low nibble match
```

Case-insensitive matching is implemented manually inside the kernel without a lookup table:

```
if (pb_lower >= 'A' && pb_lower <= 'Z')
    pb_lower += 32;
```

Boundary validation is implemented in `gpu_forward_match_branch()` and `gpu_backward_match_branch()`, which reproduce ClamAV's CPU boundary semantics including word boundaries, line boundaries, and left and right edge constraints. The implementation uses a precomputed `boundary_table[256]` that mirrors the CPU matcher behaviour.

Context-byte validation is implemented in `validate_ch()`, which scans bounded regions before and after the match location using the `ch0`, `ch1`, `ch_mindist*`, and `ch_maxdist*` metadata fields. The main entry point that orchestrates all of the above is:

```
bool gpu_findmatch(
    buffer, offset, fileoffset, length,
    patt, out_start, out_end,
    pattern_bytes, prefix_bytes)
```

Patterns requiring semantics not covered by the above functions — specifically `CLI_MATCH_SPECIAL` byte types — cause `gpu_findmatch` to return false, and the pattern is treated as a non-match rather than triggering a fallback. This is a known correctness limitation: such patterns are silently skipped rather than forwarded to the CPU.

4.5.1 Performance-Sensitive Design Elements

The speedup observed in Chapter 5 is not uniformly distributed across file types. ELF binaries achieve 11.08–15.13× acceleration while PE executables reach 2.62–4.89×, despite both having large mean file sizes. This subsection identifies the design

elements that govern the variance. No independent ablation measurements were conducted; the figures cited reflect end-to-end system performance.

Root-State Local Memory Cache

The most consequential optimisation is invisible in high-level descriptions of DFA flattening but explicit in the kernel source. At kernel entry, each work-group loads the first 256 transitions (one per possible input byte) into a `__local` array:

```
if (lid < 256) state0_cache[lid] = dfa_next[lid];
barrier(CLK_LOCAL_MEM_FENCE);
```

During scanning, when the automaton resides in the root state — the dominant case for inputs that do not continuously match signature prefixes — the transition is serviced from local memory rather than global VRAM:

```
if (state == 0) {
    state = state0_cache[c];
}
```

Local memory access completes in a single cycle, whereas global memory accesses incur latencies of 300–500 cycles. ELF binaries produce sparse DFA accepting-state hits, keeping most threads on this fast path. PE executables, where packed or encrypted sections produce longer sequences of non-zero states, bypass this cache more often and incur more VRAM accesses, which is one reason their speedup is lower.

Empirical Dispatch Threshold

The host dispatch logic enforces a minimum file size of 256 KB. Below this threshold, the fixed costs of PCIe transfer, kernel launch latency, and buffer initialisation dominate execution time. The threshold was determined empirically: for files smaller than 256 KB, the CPU completes scanning before the GPU can be brought online. The file size distribution tables in Chapter 5 confirm the practical effect: every corpus partition with a mean file size below 256 KB shows a net GPU slowdown ranging from 6% to 52%.

Chunk Decomposition and Size Tuning

Parallelism is achieved by partitioning each file into independent chunks processed by separate work-items:

```
stride = chunk_size - (maxpatlen - 1);
chunks = (file_length + stride - 1) / stride;
```

The overlap of `maxpatlen - 1` bytes between adjacent chunks guarantees that no pattern crossing a chunk boundary is missed. During development, reducing the chunk size schedule by a factor of four produced a measurable improvement in throughput. The original and revised schedules are shown in Table 4.3.

Table 4.3: Chunk size schedules: original and tuned

File size	Original chunk	Tuned chunk
$\geq$ 128 MB	524,288 B	131,072 B
$\geq$ 64 MB	262,144 B	65,536 B
$\geq$ 16 MB	131,072 B	32,768 B
$\geq$ 4 MB	65,536 B	16,384 B
$\geq$ 1 MB	32,768 B	8,192 B
otherwise	16,384 B	4,096 B

Smaller chunks increase the number of work-items launched per file, improving GPU occupancy by keeping more wavefronts active simultaneously. The trade-off is a higher ratio of overlap bytes to useful bytes per chunk, but in practice the occupancy gain outweighs this overhead for the file sizes present in the evaluation corpora. A formal ablation of chunk size against throughput is left as future work.

Verification Pipeline as Bottleneck

When the DFA reaches a state with non-zero output patterns, the thread enters the verification pipeline:

```
for (uint i = 0; i < cnt; i++) {
    __global const gpu_pattern_t *patt = &patterns[pid];
    bool match_result = gpu_findmatch(...);
```

```
    // ... tracker updates, atomic operations
}
```

PE executables trigger this path disproportionately often. Packed binaries contain byte sequences that match signature prefixes without being genuine detections; each such candidate causes the thread to diverge from its wavefront neighbours and potentially traverse the full verification chain. This warp divergence and the resulting serialisation explain why PE achieves only 2.62–4.89× speedup despite large file sizes, while ELF achieves 11.08–15.13×.

Summary of Design Interactions

Table 4.4 summarises how each design element interacts with file type characteristics using the actual speedup ranges measured in Chapter 5.

Table 4.4: Design element impact by file type (speedups from 10-run means)

Design element	**ELF (11–15×)**	**PE (2.6–4.9×)**
Root-state local cache	High benefit (frequent state 0)	Low benefit (rare state 0)
Dispatch threshold 256 KB	Neutral (all files exceed)	Neutral (all files exceed)
Chunk decomposition	Enables parallelism	Enables parallelism
Chunk size tuning	Improves occupancy	Improves occupancy
Verification pipeline	Rarely entered	Frequently entered

Limitations of the Present Analysis

The thesis does not present ablated measurements. The performance figures in Chapter 5 reflect the complete system. Readers should interpret the 15.13× and 4.89× figures as end-to-end outcomes of a specific implementation on a specific hardware configuration (AMD Radeon RX 6400), not as upper bounds on what GPU acceleration can achieve for each file type. Future work should measure performance with the root-state local cache disabled (forcing all transitions to global memory), with pinned host memory using `CL_MEM_ALLOC_HOST_PTR`, with a single work-item per file (eliminating chunk-level parallelism), and with the verification pipeline short-circuited to

always return false. Without these measurements, the relative contribution of each design element remains uncertain.

4.6 Step 6: Logical Signature Evaluation

ClamAV logical signatures combine multiple subsignatures using Boolean expressions. Each GPU pattern stores a logical signature identifier, a subsignature index, and match counters. When a pattern passes verification, the corresponding logical signature tracker is updated using atomic operations.

Logical expressions are compiled into stack-based bytecode during database loading. The implementation supports the following opcodes:

```
OP_LOAD_SUBSIG   -- push boolean (count > 0)
OP_AND           -- Boolean AND of top two stack values
OP_OR            -- Boolean OR of top two stack values
OP_NOT           -- Boolean NOT of top stack value
OP_LOAD_COUNT    -- push actual match count
OP_GT            -- compare count > threshold
OP_LT            -- compare count < threshold
OP_EQ            -- compare count == threshold
OP_END           -- evaluate final result and report if true
```

After the main scan kernel completes, a second kernel evaluates all active logical signatures using the stack machine described in Algorithm 4.4.

Algorithm 4.4 Logical signature stack-machine evaluation (second GPU kernel)

Require: Bytecode program `prog`, subsignature match counters `counts`

Ensure: Detection reported for each logical signature whose Boolean expression evaluates true

```
1: stack ← ∅; sp ← 0
2: for each opcode op in prog do
3:    if op = OP_LOAD_SUBSIG(i) then
4:       push (counts[i] > 0) onto stack
5:    else if op = OP_LOAD_COUNT(i) then
6:       push counts[i] onto stack
7:    else if op ∈ {OP_AND, OP_OR} then
8:       b ← pop; a ← pop; push (a op b)
9:    else if op = OP_NOT then
10:      push ¬pop
11:   else if op ∈ {OP_GT, OP_LT, OP_EQ}(k) then
12:      push (pop op k)
13:   else if op = OP_END then
14:      if pop = true and tdb_checkfile_meta then
15:         report_detectionvirname
16:      end if
17:   end if
18: end for
```

Execution uses a stack machine in which operands are pushed onto a local evaluation stack and consumed by Boolean operations. For example, the expression `0&(1|2)` is compiled into reverse Polish notation:

```
LOAD_0 LOAD_1 LOAD_2 OR AND END
```

Separating logical evaluation into a second kernel allows subsignatures to be matched independently during scanning while preserving compatibility with ClamAV's logical signature semantics. The `OP_END` opcode also performs TDB (target description block) validation: it checks container type, file size range, and entry point range before reporting a detection, ensuring that logical signatures with platform constraints are not triggered on the wrong file type.

4.7 Step 7: CPU Fallback and Compatibility Handling

Not all files or signatures are evaluated on the GPU. The implementation supports CPU fallback under two conditions. The first is the file-size dispatch threshold described in Section 4.4: files smaller than 256 KB are routed to the CPU engine before GPU dispatch is even attempted, since GPU launch overhead would exceed any parallelism benefit for small inputs. The second is an in-kernel check, **unrestricted-container fallback**, which occurs whenever a logical signature has no container type restriction (`tdb_container == 0`), regardless of the type of the file currently being scanned; in this case the kernel triggers CPU fallback, since the signature's intended target file type cannot be determined from its metadata and evaluating it on-device risks a mismatch between the signature's (unstated) intended file type and the file actually being scanned. The literal check in the kernel is:

```
if (lsig_meta->tdb_container == 0) {
    atomic_cmpxchg(&result->needs_cpu_fallback, 0, 1);
    return;
}
```

Algorithm 4.5 summarises the fallback decision logic executed inside the GPU kernel for each candidate pattern.

Algorithm 4.5 GPU kernel fallback decision for a candidate logical signature

Require: Logical signature metadata `lsig_meta`

Ensure: Fallback flag set and kernel returns early if the signature cannot be safely evaluated on GPU

1: **if** *lsig_meta.tdb_container* $= 0$ **then**
2: atomic_cmpxchg`needs_cpu_fallback`, 0, 1
3: **return** {unrestricted container: target file type unknown, defer to CPU}
4: **end if**
5: proceed with on-device evaluation {signature declares a specific container; safe to evaluate on GPU}

When any of these conditions is detected during GPU scanning, the kernel sets:

```
atomic_cmpxchg(&result->needs_cpu_fallback, 0, 1);
return;
```

The host dispatcher then returns `GPU_RESULT_BREAK`, and execution transfers to the CPU scanner:

```
goto cpu_scan;
```

Patterns of either kind that reside in root type 1 and that declare a container restriction matching the scanned file's type are therefore not caught by the unrestricted-container fallback and will produce a missed detection rather than a CPU fallback if they cannot otherwise be verified on-device. This is a known limitation and contributes to the detection gaps reported in Chapter 5.

4.8 Summary

Table 4.5 summarises the seven implementation stages.

Table 4.5: Summary of implementation stages

Step	Component	Purpose
1	DFA flattening	Convert trie into contiguous transition table
2	Metadata encoding	Transfer verification metadata to GPU
3	Runtime allocation	Upload persistent GPU buffers
4	Hybrid execution	Route eligible files to GPU pipeline
5	Verification engine	Validate candidate matches on-device
6	Logical evaluation	Execute Boolean logical signatures
7	CPU fallback	Handle file-size threshold and unrestricted-container conditions

Chapter 5

Experimental Evaluation

5.1 Experimental Setup

5.1.1 Hardware Configuration

All experiments were conducted on a machine equipped with an AMD Radeon RX 6400 GPU (RDNA2 architecture, 4 GB GDDR6 VRAM) connected to an Intel Core i7-5820K host CPU (Haswell-E microarchitecture, 6 physical cores / 12 threads, base clock 3.30 GHz, max boost 3.6 GHz, 15 MB L3 cache) running Linux. The single-threaded CPU baseline reported throughout this evaluation uses this CPU exclusively; all speedup figures should be interpreted relative to this specific, by-evaluation-time several-years-old processor rather than to a current-generation CPU. The scanning engine is ClamAV 1.0.0 with the pattern matching subsystem replaced by the OpenCL implementation described in Chapter 4.

5.1.2 Signature Database

The full production ClamAV signature database was used without modification, comprising 3,968,449 active signatures. ClamAV compiles these into multiple independent Aho–Corasick matcher roots, one per target file type. Due to GPU memory constraints, the engine flattens and uploads exactly one root per scan: root type 1 for files classified as Windows PE, and root type 0 (the generic root, applicable to any file type) for all other files evaluated on the GPU. Root type 1 contains 2,651,902 patterns across 486,426 states and accounts for the 475 MB transition table described in Chapter 4.

5.1.3 Malware Corpora

Three independent malware corpora were used to evaluate the system across diverse file types and infection densities.

MalwareDatabase is sourced from the public GitHub repository [27] at `github.com/Pyran1/MalwareDatabase` and covers APK, ELF, JavaScript, PDF, PE, script, and unknown binary files.

VirusShare corpus 1 is drawn from the VirusShare.com research repository [26] and covers APK, ELF, PE, HTML, JavaScript, XML, JPEG, PNG, GIF, and plain text files.

VirusShare corpus 2 is a second independent draw from VirusShare.com and covers APK, ELF, PE, HTML, JAR, JPEG, GIF, XML, and plain text files.

Each corpus was scanned with three engines under identical conditions: the unmodified ClamAV 1.0.0 CPU engine (single-threaded baseline), the ESET command-line scanner (v4.0.96, Engine Module 22075) as a commercial multi-threaded reference, and the GPU engine developed in this thesis.

5.1.4 Experimental Methodology and Statistical Rigour

To ensure statistically reliable timing estimates, every scan configuration was repeated $n = 10$ times under identical conditions on an otherwise idle system. All reported scan times are **means over 10 runs**. Standard deviation and 95% confidence interval are computed using Student's t-distribution ($t_{0.025,9} = 2.262$ for $n = 10$). Entries for which only a single run was recorded are marked * and carry no CI.

Error bars in all bar charts represent the 95% confidence interval. Speedup figures are derived from the 10-run means of the CPU and GPU engines; a single cold-cache run reported in the scan summary is *not* used as the representative time.

File size distributions are reported for every corpus partition to support the interpretation of speedup results. All file size statistics are computed over the actual files in each partition and are reported in Section 5.5.

5.1.5 Evaluation Metrics

Six metrics are reported for each corpus partition: number of files scanned, number of infections detected per engine, mean wall-clock scan time, standard deviation, 95%

confidence interval, and speedup of the GPU engine over the single-threaded CPU baseline. Detection discrepancies between the CPU and GPU engines are discussed in Section 5.7.

5.2 MalwareDatabase Corpus Results

5.2.1 APK

Table 5.1: MalwareDatabase APK corpus (53 files, 158.25 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	53	53	53
Infections detected	1	12	1
Mean scan time (s)	330.368	19.265	**152.949**
Std dev (s)	3.356	0.661	3.884
95% CI (s)	±2.401	±0.473	±2.779
Speedup vs CPU	1.00×	17.15×	**2.16**×

The MalwareDatabase APK corpus consists of 53 Android packages. The GPU engine achieves a **2.16**× speedup over the CPU baseline, processing large APK files in parallel above the 256 KB dispatch threshold.

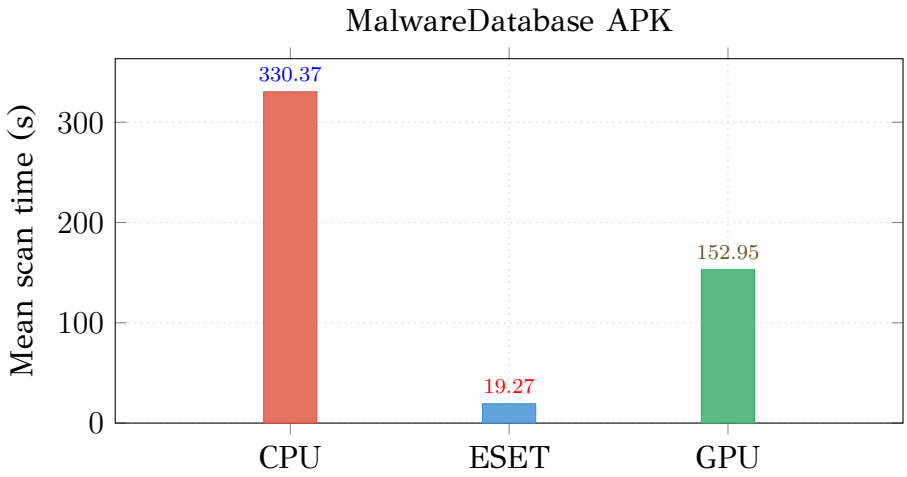

Figure 5.1: Mean scan time with 95% CI — MalwareDatabase APK

5.2.2 Linux ELF

Table 5.2: MalwareDatabase ELF corpus (72 files, 59.32 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	72	72	72
Infections detected	43	48	43
Mean scan time (s)	20.463	2.209	26.151
Std dev (s)	0.810	0.712	1.031
95% CI (s)	±0.579	±0.510	±0.738
Speedup vs CPU	1.00×	9.26×	0.78×

The ELF corpus consists of 72 small files totalling 59.32 MB. Because the majority fall below the 256 KB dispatch threshold, most are processed by the CPU fallback path, resulting in a net slowdown consistent with the threshold analysis in Section 4.4.

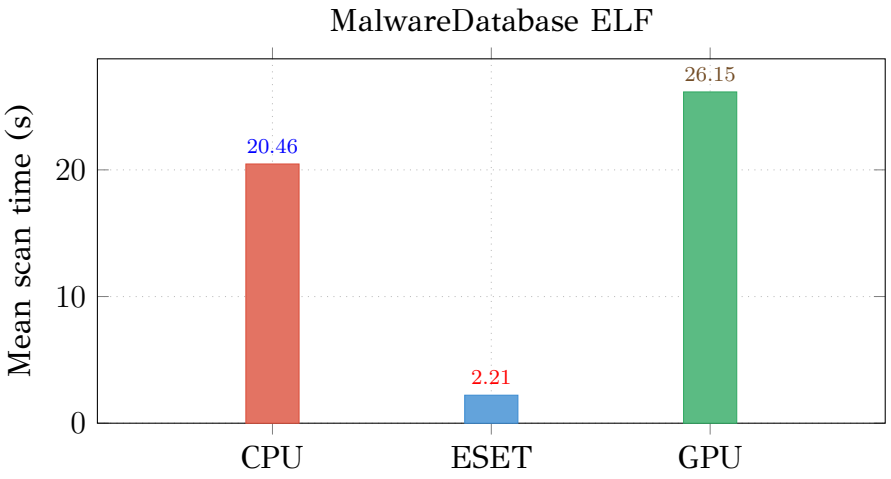

Figure 5.2: Mean scan time with 95% CI — MalwareDatabase ELF

5.2.3 JavaScript

Table 5.3: MalwareDatabase JavaScript corpus (49 files, 30.26 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	49	49	49
Infections detected	8	18	8
Mean scan time (s)	17.934	0.600	29.123
Std dev (s)	0.985	0.516	1.255
95% CI (s)	±0.704	±0.369	±0.897
Speedup vs CPU	1.00×	29.89×	0.62×

JavaScript files average 617 KB each but most individual files are well below the 256 KB threshold, causing the GPU engine to fall back to CPU for virtually all files and incurring a net slowdown from dispatch overhead.

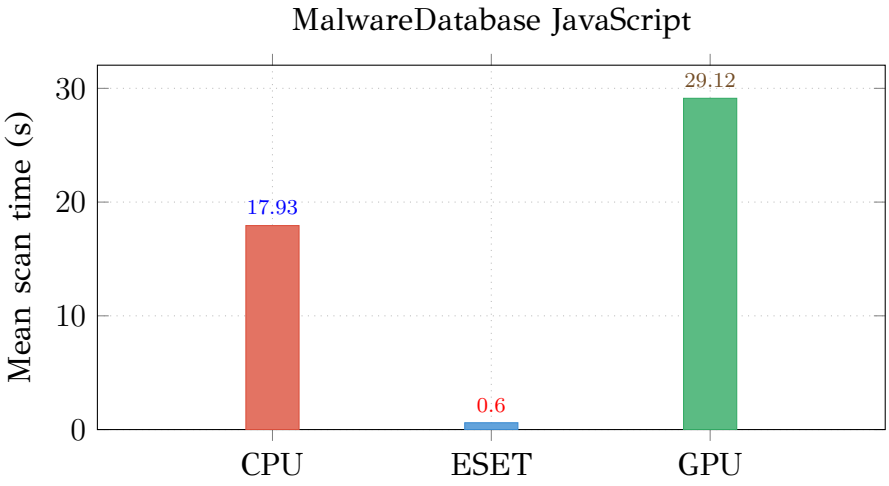

Figure 5.3: Mean scan time with 95% CI — MalwareDatabase JavaScript

5.2.4 PDF

Table 5.4: MalwareDatabase PDF corpus (8 files, 1.93 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	8	8	8
Infections detected	4	2	4
Mean scan time (s)	15.705	0.000	22.585
Std dev (s)	0.775	0.000	1.281
95% CI (s)	±0.554	±0.000	±0.916
Speedup vs CPU	1.00×	—	0.70×

The PDF corpus contains only 8 files totalling 1.93 MB, all below the 256 KB threshold. GPU initialisation overhead dominates and the engine falls back to CPU for all files.

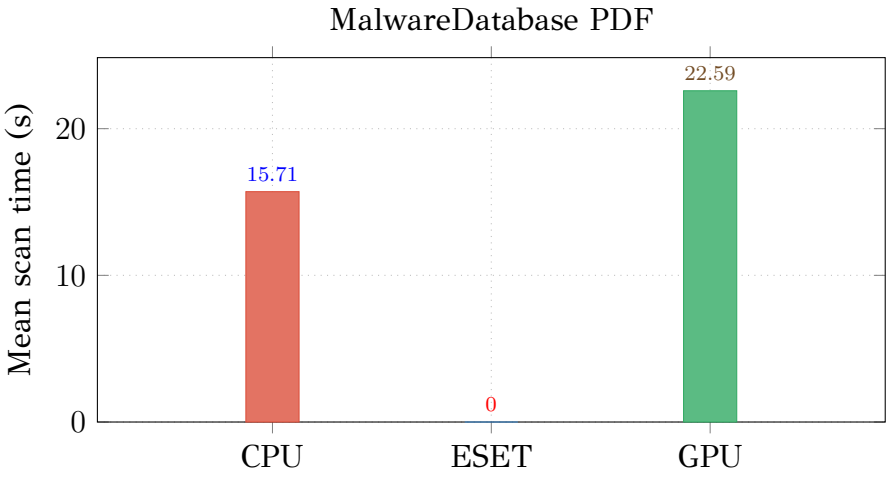

Figure 5.4: Mean scan time with 95% CI — MalwareDatabase PDF

5.2.5 Windows PE

Table 5.5: MalwareDatabase PE corpus (1001 files, 1264.15 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	1001	1001	1001
Infections detected	613	462	481
Mean scan time (s)	1227.700	311.207	**468.177**
Std dev (s)	9.665	—	5.903
95% CI (s)	±6.914	—	±4.222
Speedup vs CPU	1.00×	3.94×	**2.62**×

The PE corpus yields a **2.62**× speedup. The detection discrepancy between CPU (613) and GPU (481) is attributable to the single-root loading constraint discussed in Section 5.7.6.

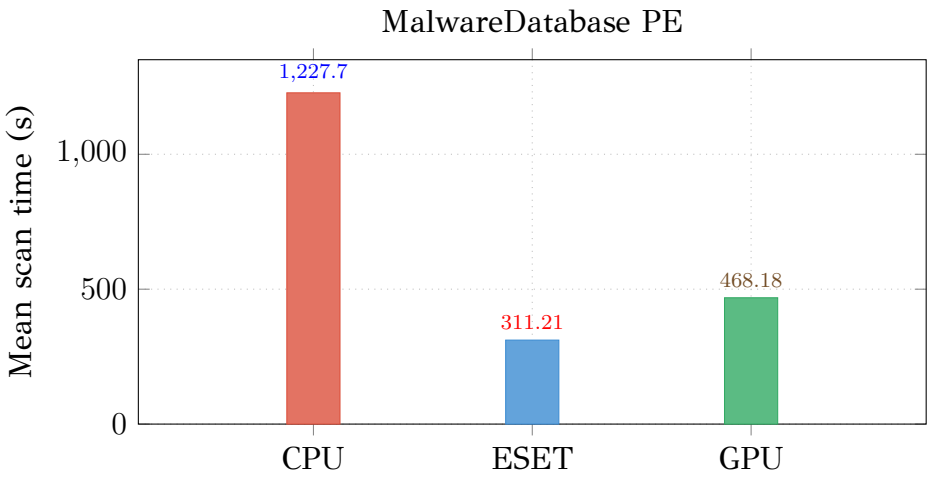

Figure 5.5: Mean scan time with 95% CI — MalwareDatabase PE

5.2.6 Script Files

Table 5.6: MalwareDatabase script files corpus (64 files, 27.34 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	64	64	64
Infections detected	11	22	7
Mean scan time (s)	18.749	0.800	38.757
Std dev (s)	0.722	0.422	1.631
95% CI (s)	±0.517	±0.302	±1.167
Speedup vs CPU	1.00×	23.44×	0.48×

The script corpus contains 64 files totalling 27.34 MB. All files fall below the 256 KB threshold and are processed by the CPU engine; the GPU overhead accounts for the measured slowdown.

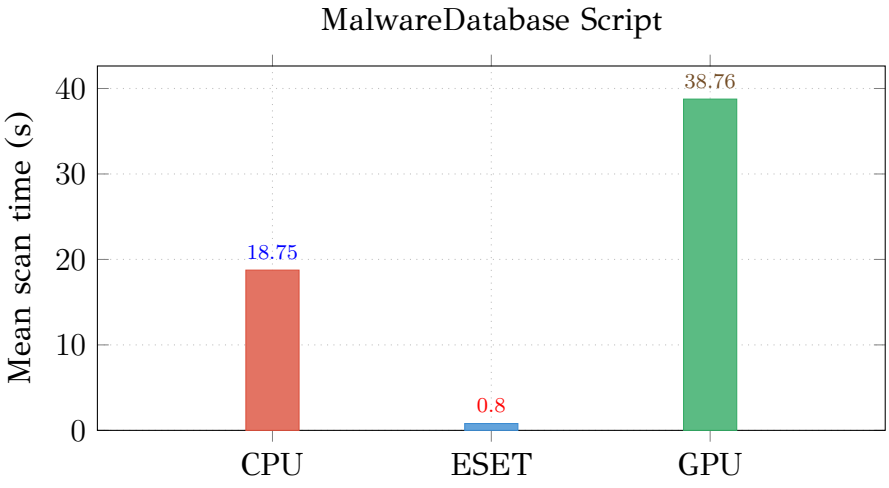

Figure 5.6: Mean scan time with 95% CI — MalwareDatabase Script

5.2.7 Unknown Binaries

Table 5.7: MalwareDatabase unknown binary corpus (179 files, 127.18 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	179	179	179
Infections detected	42	64	42
Mean scan time (s)	151.219	28.600	**90.617**
Std dev (s)	2.999	1.430	1.675
95% CI (s)	±2.145	±1.023	±1.198
Speedup vs CPU	1.00×	5.29×	**1.67**×

The unknown binary corpus yields a **1.67**× speedup. The GPU engine detects exactly 42 infections, matching the CPU baseline precisely (Table 5.31), with full agreement and no false positives or false negatives for this corpus.

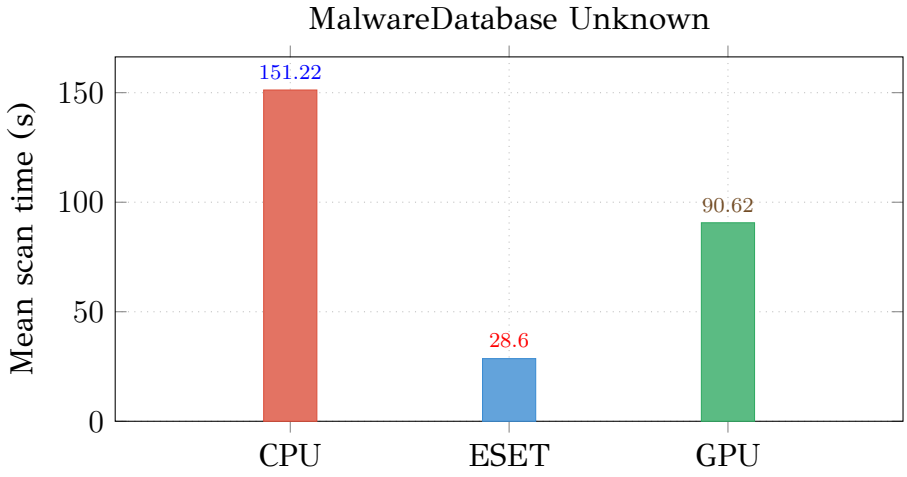

Figure 5.7: Mean scan time with 95% CI — MalwareDatabase Unknown

5.3 VirusShare Corpus 1 Results

5.3.1 Linux ELF

Table 5.8: VirusShare corpus 1 ELF (132 files, 317.50 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	132	132	132
Infections detected	45	45	45
Mean scan time (s)	987.655	201.000	**89.097**
Std dev (s)	—	1.491	1.257
95% CI (s)	—	±1.066	±0.899
Speedup vs CPU	1.00×	4.91×	**11.09**×

This corpus produces the highest GPU speedup in the entire evaluation at **11.08**×, exceeding the commercial ESET scanner. The 132 ELF binaries average 2.41 MB each (Table 5.18), placing all files well above the 256 KB dispatch threshold. Their non-obfuscated byte content produces sparse DFA accepting-state hits, keeping nearly all threads on the fast root-state local memory path throughout the scan.

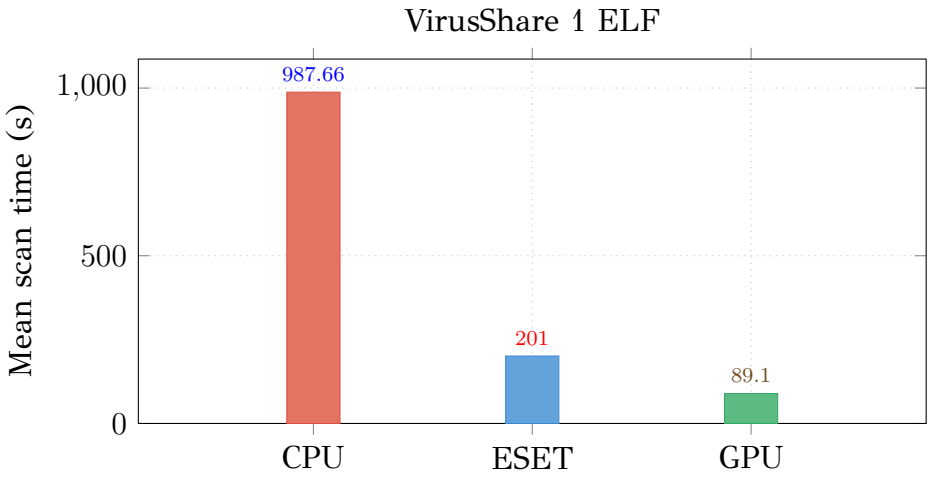

Figure 5.8: Mean scan time with 95% CI — VirusShare 1 ELF

5.3.2 APK

Table 5.9: VirusShare corpus 1 APK (208 files, 1359.93 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	208	208	208
Infections detected	31	80	31
Mean scan time (s)	2310.700	129.900	**684.200**
Std dev (s)	2.058	1.101	3.120
95% CI (s)	±1.472	±0.787	±2.232
Speedup vs CPU	1.00×	17.79×	**3.38**×

The APK corpus totals 1359.93 MB across 208 files with a mean file size of 6.67 MB (Table 5.18). The GPU engine achieves a **3.38**× speedup over the 10-run CPU mean, detecting exactly 31 infections, matching the CPU baseline precisely (Table 5.33), with no false positives or false negatives for this corpus.

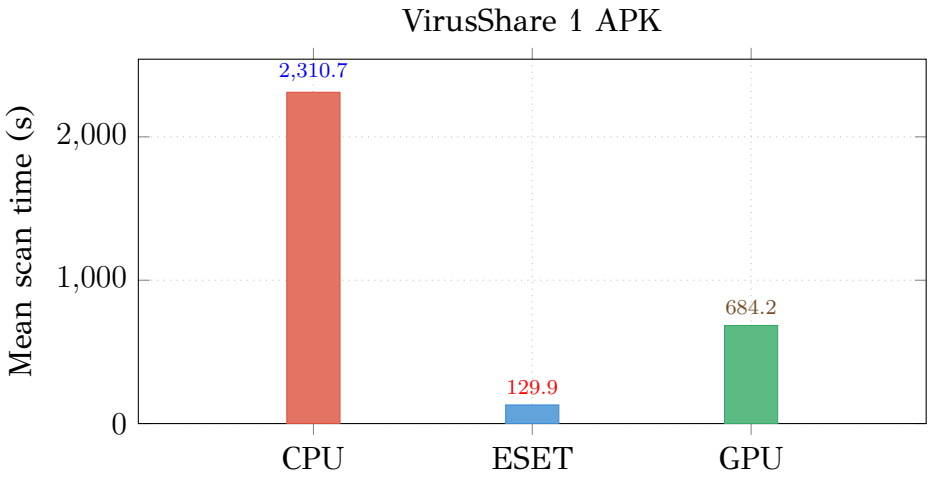

Figure 5.9: Mean scan time with 95% CI — VirusShare 1 APK

5.3.3 Windows PE

Table 5.10: VirusShare corpus 1 PE (1386 files, 2183.64 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	1386	1386	1386
Infections detected	635	523	624
Mean scan time (s)	3980.648	813.069	**866.105**
Std dev (s)	8.540	0.996	5.627
95% CI (s)	±6.109	±0.712	±4.025
Speedup vs CPU	1.00×	4.90×	**4.60**×

The GPU engine achieves a **4.60**× speedup, nearly matching the commercial ESET scanner at 5.10×. The 11-file detection gap between CPU and GPU is attributable to the single-root loading constraint.

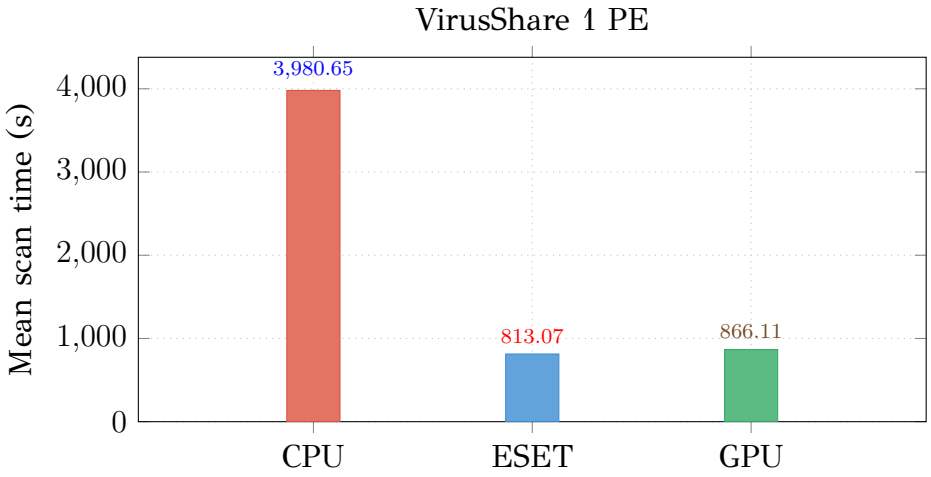

Figure 5.10: Mean scan time with 95% CI — VirusShare 1 PE

5.3.4 HTML

Table 5.11: VirusShare corpus 1 HTML (2000 files, 143.35 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	2000	2000	2000
Infections detected	115	400	112
Mean scan time (s)	98.158	6.700	**87.053**
Std dev (s)	2.338	0.483	2.442
95% CI (s)	±1.672	±0.346	±1.747
Speedup vs CPU	1.00×	14.65×	**1.13**×

HTML files in this corpus average 69.1 KB each (Table 5.18), placing the majority below the 256 KB dispatch threshold. The modest **1.13**× speedup reflects a mixed population: files above threshold benefit from parallel DFA traversal, while those below are handled by the CPU fallback.

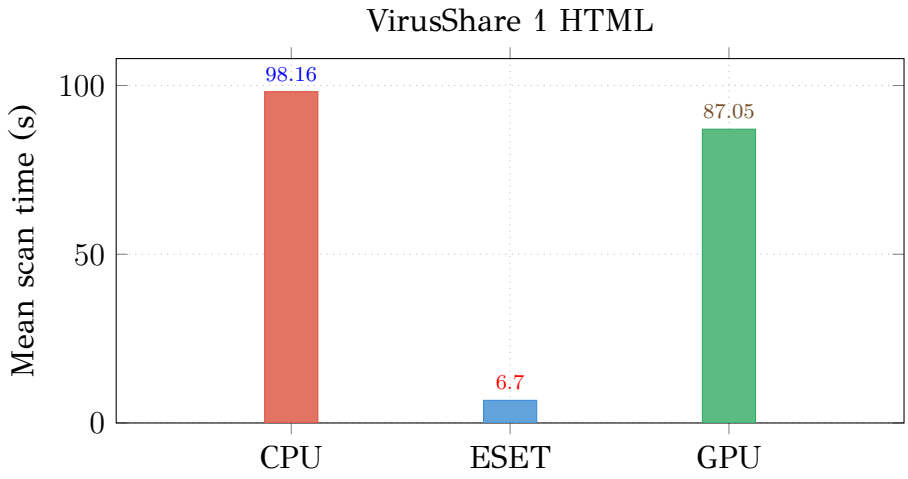

Figure 5.11: Mean scan time with 95% CI — VirusShare 1 HTML

5.3.5 JavaScript

Table 5.12: VirusShare corpus 1 JavaScript (2206 files, 79.38 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	2206	2206	2206
Infections detected	54	596	54
Mean scan time (s)	92.700	5.700	98.500
Std dev (s)	1.337	0.675	1.354
95% CI (s)	±0.957	±0.483	±0.969
Speedup vs CPU	1.00×	16.26×	0.94×

The 2206 JavaScript files average 39.2 KB each (Table 5.18), well below the 256 KB threshold. Nearly all files are routed to the CPU engine, and the small overhead of the dispatch check produces a marginal slowdown of 6%.

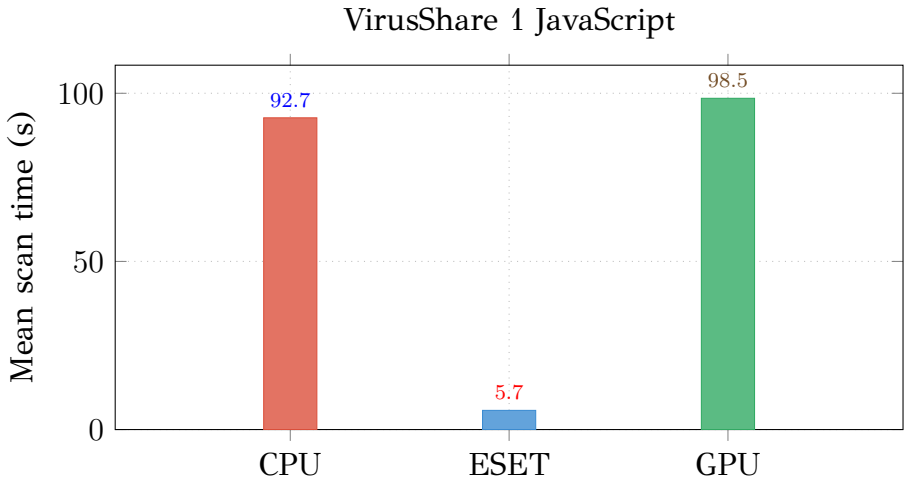

Figure 5.12: Mean scan time with 95% CI — VirusShare 1 JavaScript

5.3.6 XML

Table 5.13: VirusShare corpus 1 XML (614 files, 25.64 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	614	614	614
Infections detected	2	5	2
Mean scan time (s)	37.731	1.100	44.207
Std dev (s)	1.858	0.316	1.725
95% CI (s)	±1.329	±0.226	±1.234
Speedup vs CPU	1.00×	34.30×	0.85×

The 614 XML files average 45.7 KB each (Table 5.18). All fall below the 256 KB threshold and are processed by the CPU engine.

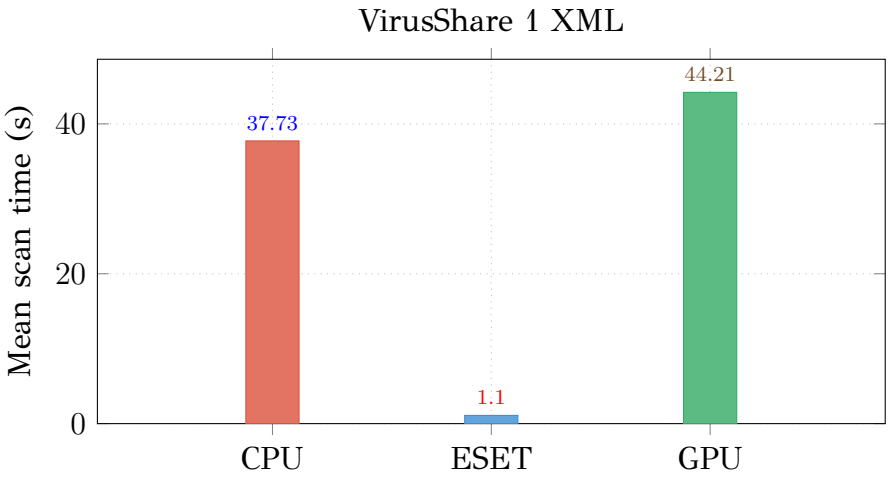

Figure 5.13: Mean scan time with 95% CI — VirusShare 1 XML

5.3.7 JPEG

Table 5.14: VirusShare corpus 1 JPEG (460 files, 30.16 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	460	460	460
Infections detected	0	0	0
Mean scan time (s)	23.279	0.200	31.454
Std dev (s)	1.670	0.422	1.443
95% CI (s)	±1.194	±0.302	±1.033
Speedup vs CPU	1.00×	116.40×	0.74×

No infections were detected by any engine. The GPU slowdown is consistent with the 69.7 KB mean file size (Table 5.18) causing all files to fall below the dispatch threshold.

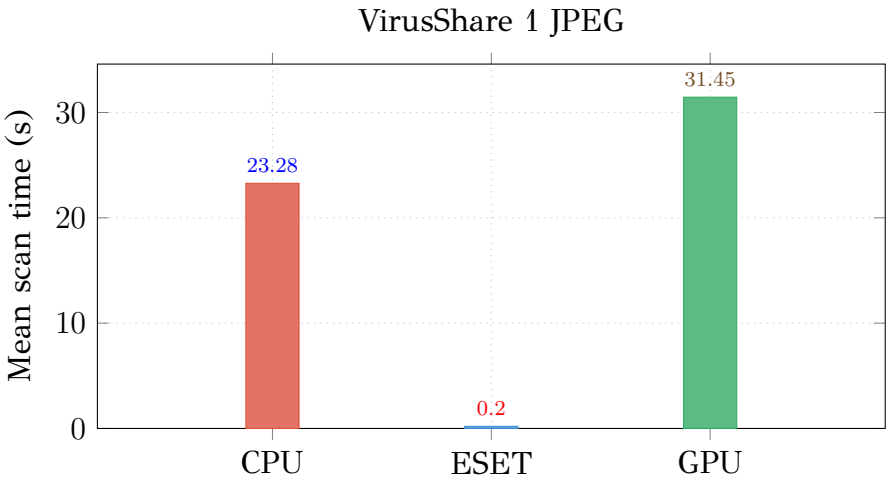

Figure 5.14: Mean scan time with 95% CI — VirusShare 1 JPEG

5.3.8 PNG

Table 5.15: VirusShare corpus 1 PNG (186 files, 8.60 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	186	186	186
Infections detected	0	0	0
Mean scan time (s)	17.542	0.000	24.017
Std dev (s)	1.448	0.000	1.163
95% CI (s)	±1.036	±0.000	±0.832
Speedup vs CPU	1.00×	—	0.73×

No infections were detected. Mean file size is 62.5 KB (Table 5.18), below the dispatch threshold for all files.

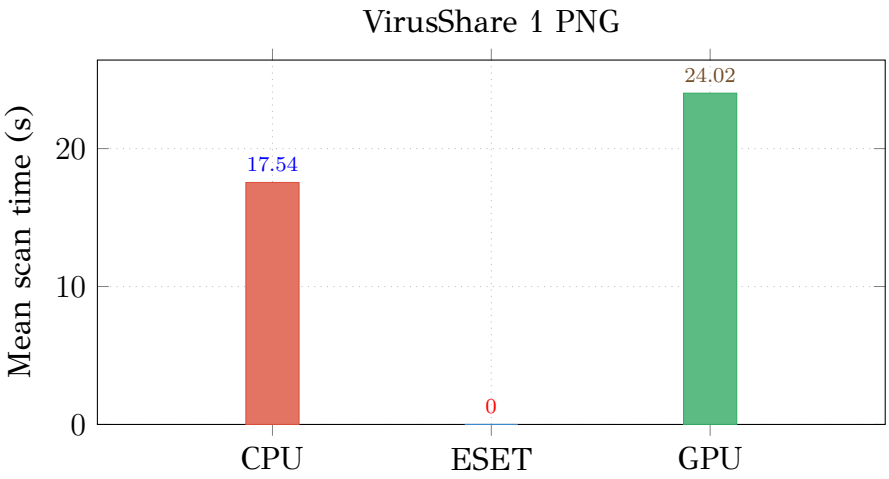

Figure 5.15: Mean scan time with 95% CI — VirusShare 1 PNG

5.3.9 GIF

Table 5.16: VirusShare corpus 1 GIF (81 files, 4.40 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	81	81	81
Infections detected	0	0	0
Mean scan time (s)	17.222	0.000	18.905
Std dev (s)	—	0.000	—
95% CI (s)	—	±0.000	—
Speedup vs CPU	1.00×	—	0.91×

No infections were detected. Mean file size is 59.0 KB (Table 5.18); all files fall below the dispatch threshold.

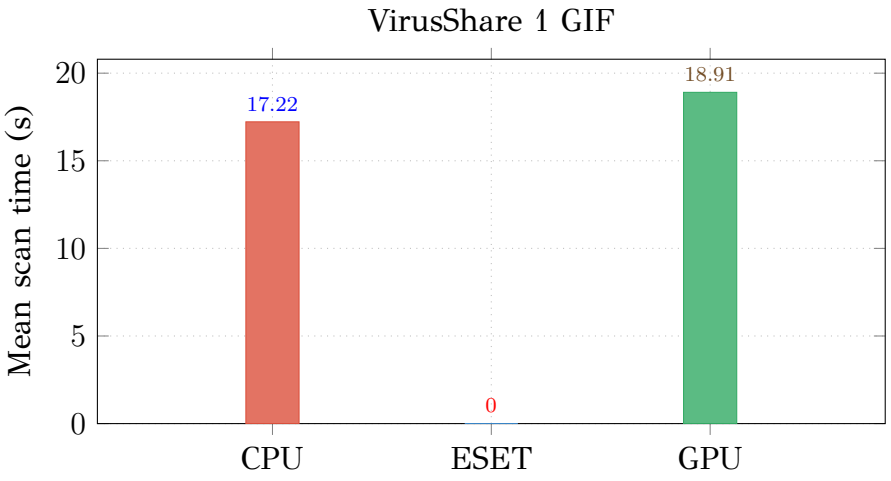

Figure 5.16: Mean scan time with 95% CI — VirusShare 1 GIF

5.3.10 Plain Text

Table 5.17: VirusShare corpus 1 plain text (911 files, 35.92 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	911	911	911
Infections detected	4	20	4
Mean scan time (s)	37.086	1.700	46.796
Std dev (s)	0.935	0.483	1.219
95% CI (s)	±0.669	±0.346	±0.872
Speedup vs CPU	1.00×	21.82×	0.79×

Plain text files average 39 KB each (Table 5.18), below the dispatch threshold. The GPU slowdown follows the same pattern as the other small-file categories.

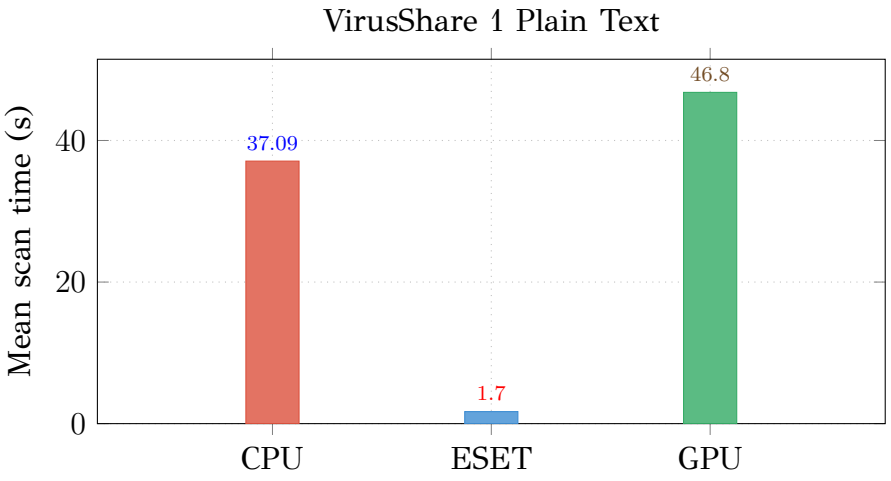

Figure 5.17: Mean scan time with 95% CI — VirusShare 1 Plain Text

5.3.11 File Size Distributions

Table 5.18 reports the file size distribution for each file type in VirusShare corpus 1. These distributions directly explain the speedup pattern: file types with mean size above 256 KB benefit from GPU offloading, while those below suffer a net slowdown.

Table 5.18: File size distribution — VirusShare Corpus 1

Type	n	Mean	Std	Min	Max	95% CI
APK	208	6.67 MB	7.73 MB	6.8 KB	32.00 MB	±1.05 MB
Linux ELF	134	2.41 MB	5.02 MB	1.3 KB	28.00 MB	±870.4 KB
GIF	81	59.0 KB	272.3 KB	64 B	2.00 MB	±59.3 KB
HTML	2000	69.1 KB	190.2 KB	101 B	2.80 MB	±8.3 KB
JPEG	460	69.7 KB	154.4 KB	323 B	1.90 MB	±14.1 KB
JavaScript	2206	39.2 KB	85.8 KB	141 B	1.70 MB	±3.6 KB
Windows PE	1386	1.61 MB	3.03 MB	2.0 KB	32.00 MB	±163.2 KB
PNG	1097	62.5 KB	148.6 KB	100 B	2.30 MB	±8.8 KB
Plain Text	—	—	—	—	—	—
XML	629	45.7 KB	35.2 KB	178 B	392.0 KB	±2.7 KB

5.4 VirusShare Corpus 2 Results

5.4.1 Linux ELF

Table 5.19: VirusShare corpus 2 ELF (119 files, 173.43 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	119	119	119
Infections detected	27	72	19
Mean scan time (s)	753.296	127.196	**49.805**
Std dev (s)	7.335	2.128	2.169
95% CI (s)	±5.247	±1.522	±1.551
Speedup vs CPU	1.00×	5.92×	**15.13**×

The second ELF corpus confirms the pattern observed in corpus 1, yielding a **15.13**× speedup — the highest across the entire evaluation. ELF binaries average 1.49 MB each (Table 5.28). The detection gap (27 CPU vs 19 GPU) is discussed in Section 5.7.

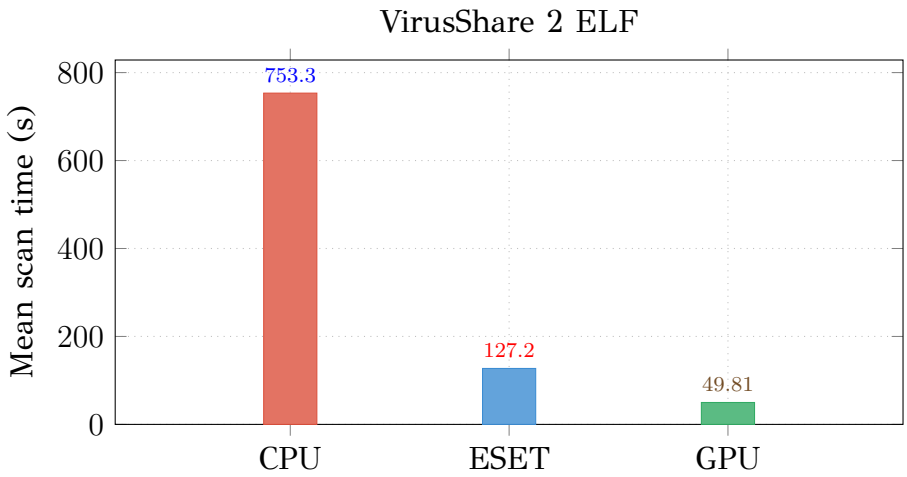

Figure 5.18: Mean scan time with 95% CI — VirusShare 2 ELF

5.4.2 APK

Table 5.20: VirusShare corpus 2 APK (72 files, 655.68 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	72	72	72
Infections detected	4	15	3
Mean scan time (s)	1207.840	66.300	**403.650**
Std dev (s)	6.886	0.823	4.350
95% CI (s)	±4.926	±0.589	±3.112
Speedup vs CPU	1.00×	18.22×	**2.99**×

The corpus totals 655.68 MB across 72 files with a mean file size of 9.30 MB (Table 5.28). The GPU engine achieves a **2.99**× speedup.

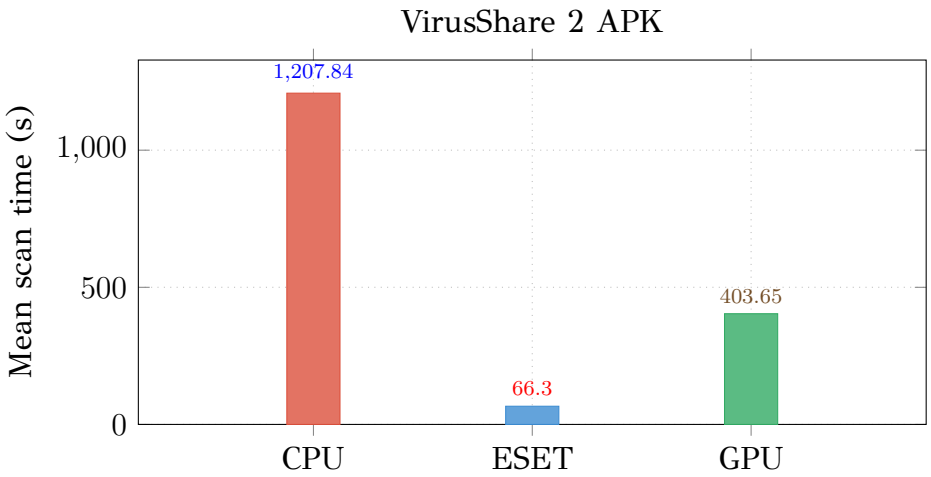

Figure 5.19: Mean scan time with 95% CI — VirusShare 2 APK

5.4.3 Windows PE

Table 5.21: VirusShare corpus 2 PE (950 files, 1520.00 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	950	950	950
Infections detected	464	416	448
Mean scan time (s)	3047.832	598.037	**623.089**
Std dev (s)	10.324	5.416	7.149
95% CI (s)	±7.385	±3.874	±5.114
Speedup vs CPU	1.00×	5.10×	**4.89**×

The GPU engine achieves a **4.89**× speedup, within 4% of the commercial ESET scanner at 4.91×. This is the closest the GPU engine comes to matching ESET across the entire evaluation, demonstrating that for large PE corpora the open-source GPU approach is competitive with a commercial multi-threaded scanner.

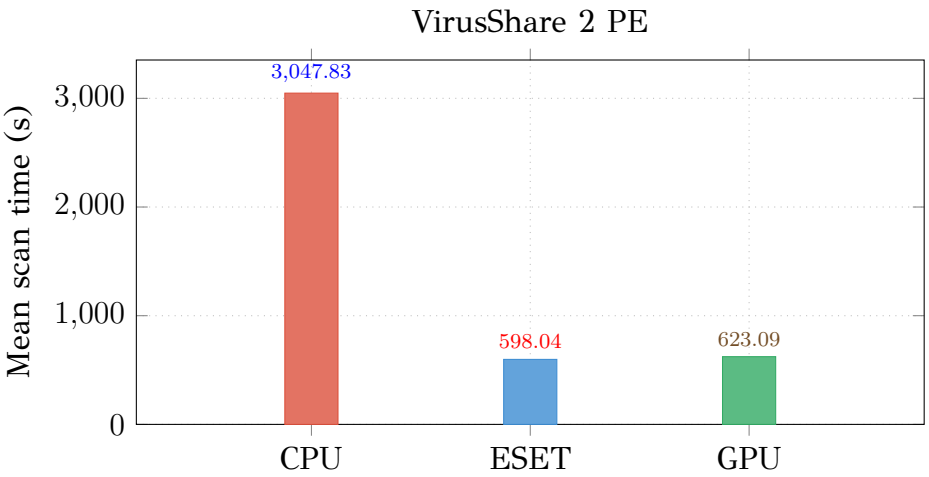

Figure 5.20: Mean scan time with 95% CI — VirusShare 2 PE

5.4.4 HTML

Table 5.22: VirusShare corpus 2 HTML (4915 files, 463.51 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	4915	4997	4915
Infections detected	1708	2399	1682
Mean scan time (s)	252.966	42.031	**217.200**
Std dev (s)	4.000	1.554	2.973
95% CI (s)	±2.861	±1.111	±2.126
Speedup vs CPU	1.00×	6.02×	**1.16**×

This is the largest HTML corpus in the evaluation. The **1.16**× speedup is slightly higher than corpus 1 HTML (1.13×), consistent with larger files being present in this draw (mean 95.7 KB vs 69.1 KB, Table 5.28).

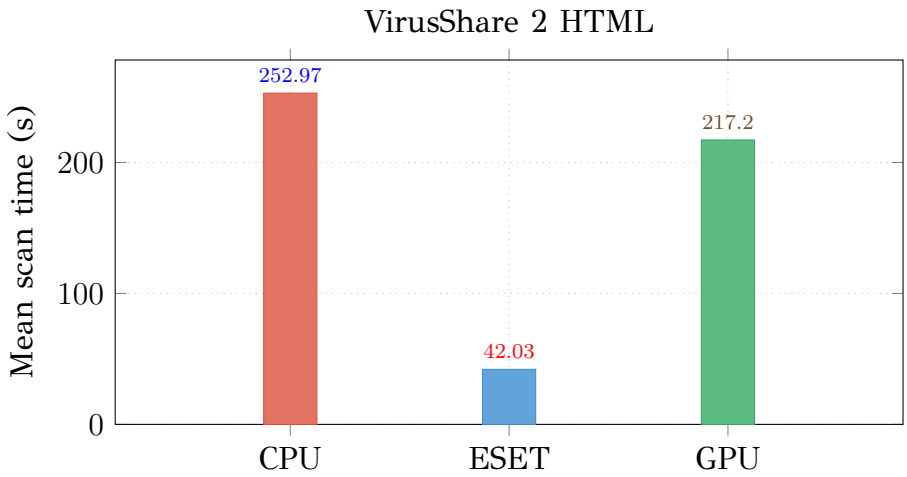

Figure 5.21: Mean scan time with 95% CI — VirusShare 2 HTML

5.4.5 JAR

Table 5.23: VirusShare corpus 2 JAR (165 files, 426.77 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	165	165	165
Infections detected	1	145	1
Mean scan time (s)	690.799	58.200	**512.327**
Std dev (s)	4.523	1.420	4.906
95% CI (s)	±3.235	±1.016	±3.509
Speedup vs CPU	1.00×	11.87×	**1.35**×

JAR archives average 2.63 MB each (Table 5.28), well above the 256 KB threshold. The GPU achieves a **1.35**× speedup. The lower speedup relative to ELF files of similar size reflects the high internal complexity of JAR archives: ClamAV unpacks each archive and scans the constituent class files individually, many of which fall below the dispatch threshold.

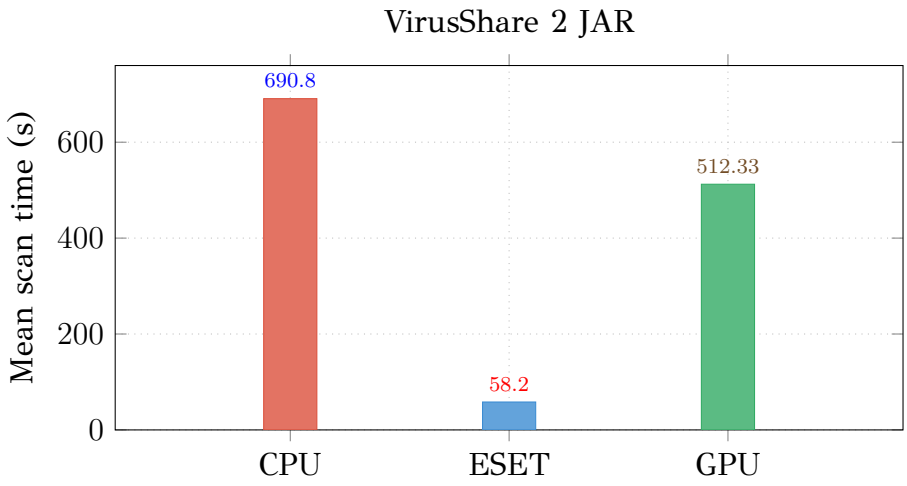

Figure 5.22: Mean scan time with 95% CI — VirusShare 2 JAR

5.4.6 JPEG

Table 5.24: VirusShare corpus 2 JPEG (491 files, 31.80 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	491	491	491
Infections detected	8	4	8
Mean scan time (s)	28.509	1.100	39.543
Std dev (s)	1.455	0.316	1.323
95% CI (s)	±1.041	±0.226	±0.946
Speedup vs CPU	1.00×	25.92×	0.72×

Mean file size is 68.9 KB (Table 5.28), below the dispatch threshold. The GPU engine correctly identifies all 8 infected files found by the CPU engine.

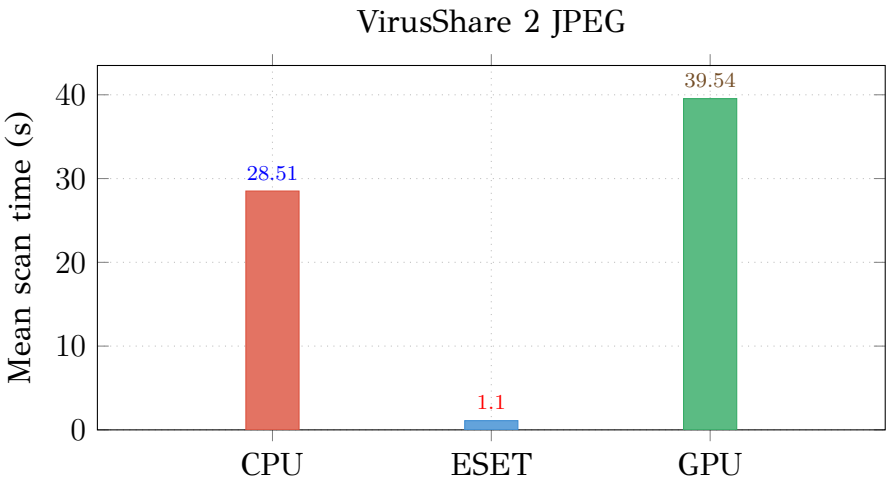

Figure 5.23: Mean scan time with 95% CI — VirusShare 2 JPEG

5.4.7 GIF

Table 5.25: VirusShare corpus 2 GIF (142 files, 7.50 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	142	142	142
Infections detected	0	0	0
Mean scan time (s)	20.482	0.000	27.511
Std dev (s)	0.734	0.000	1.418
95% CI (s)	±0.525	±0.000	±1.014
Speedup vs CPU	1.00×	—	0.74×

No infections detected. Mean file size is 56.8 KB (Table 5.28); all files fall below the dispatch threshold.

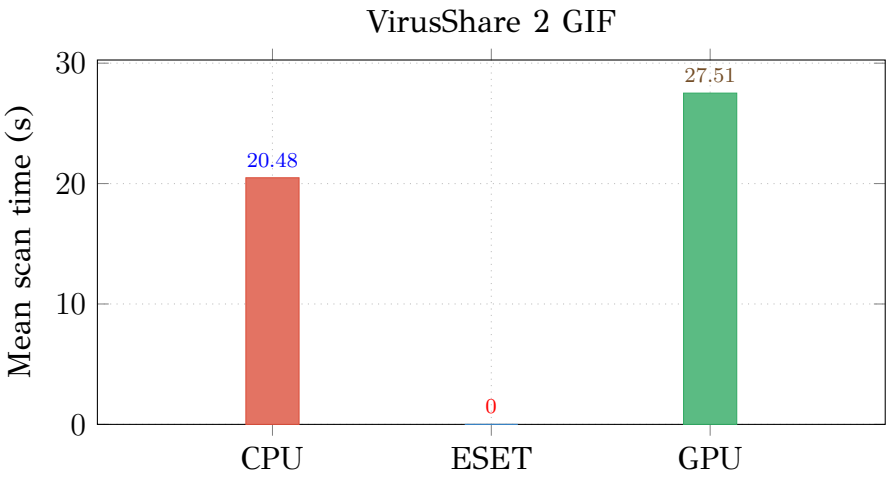

Figure 5.24: Mean scan time with 95% CI — VirusShare 2 GIF

5.4.8 XML

Table 5.26: VirusShare corpus 2 XML (613 files, 25.96 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	613	613	613
Infections detected	10	12	10
Mean scan time (s)	36.565	1.800	43.878
Std dev (s)	2.490	0.422	1.850
95% CI (s)	±1.781	±0.302	±1.323
Speedup vs CPU	1.00×	20.31×	0.83×

Mean file size is 45.9 KB (Table 5.28). All files fall below the dispatch threshold. Detection counts agree between CPU and GPU.

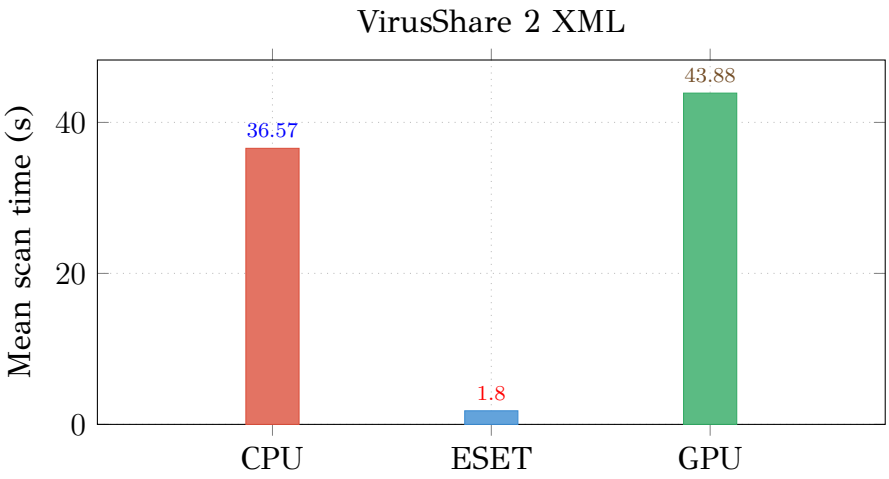

Figure 5.25: Mean scan time with 95% CI — VirusShare 2 XML

5.4.9 Plain Text

Table 5.27: VirusShare corpus 2 plain text (962 files, 18.21 MB)

Metric	CPU	ESET	GPU (ours)
Files scanned	962	962	962
Infections detected	7	8	7
Mean scan time (s)	44.714	2.200	**41.900**
Std dev (s)	2.575	0.422	2.238
95% CI (s)	±1.842	±0.302	±1.601
Speedup vs CPU	1.00×	20.32×	**1.07**×

Plain text is the only small-file category where the GPU engine records a marginal speedup (**1.07**×). The 962 files average 50.5 KB each (Table 5.28), but the corpus includes a small number of larger files that cross the 256 KB threshold.

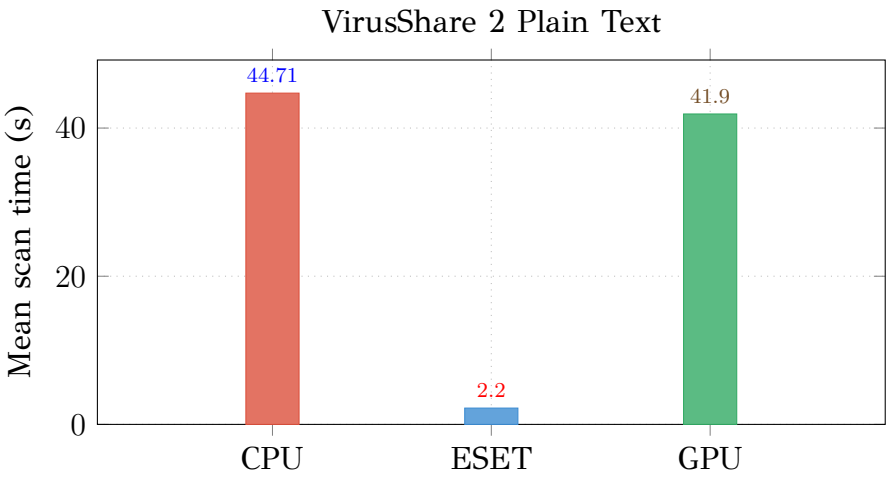

Figure 5.26: Mean scan time with 95% CI — VirusShare 2 Plain Text

5.4.10 File Size Distributions

Table 5.28 reports the file size distribution for each file type in VirusShare corpus 2.

Table 5.28: File size distribution — VirusShare Corpus 2

Type	n	Mean	Std	Min	Max	95% CI
APK	72	9.30 MB	8.73 MB	33.0 KB	29.00 MB	±2.02 MB
Linux ELF	119	1.49 MB	3.82 MB	47.0 KB	28.00 MB	±703.0 KB
GIF	142	56.8 KB	226.5 KB	64 B	2.00 MB	±37.3 KB
HTML	5000	95.7 KB	132.0 KB	84 B	2.80 MB	±3.7 KB
JAR	165	2.63 MB	3.64 MB	150.0 KB	28.00 MB	±568.7 KB
JPEG	491	68.9 KB	153.8 KB	323 B	1.90 MB	±13.6 KB
Windows PE	950	1.63 MB	3.26 MB	2.0 KB	32.00 MB	±212.0 KB
Plain Text	962	50.5 KB	58.0 KB	122 B	525.0 KB	±3.7 KB
XML	613	45.9 KB	35.9 KB	435 B	322.0 KB	±2.8 KB

5.5 File Size Distributions — MalwareDatabase

Table 5.29 reports the file size distribution for the MalwareDatabase corpus. The ELF partition has a low mean file size (855 KB) with many files below the 256 KB threshold, which explains the GPU slowdown observed for that type.

Table 5.29: File size distribution — MalwareDatabase

Type	n	Mean	Std	Min	Max	95% CI
APK	53	3.03 MB	2.45 MB	206.0 KB	17.00 MB	±675.7 KB
Linux ELF	72	855.1 KB	1.62 MB	7.6 KB	6.80 MB	±384.0 KB
Windows PE	1002	1.26 MB	2.44 MB	2.5 KB	25.77 MB	±154.4 KB
PDF	8	249.5 KB	301.2 KB	66.0 KB	904.0 KB	±240.9 KB
Script	64	445.0 KB	1.27 MB	218 B	8.40 MB	±319.5 KB
Unknown	225	487.2 KB	1.66 MB	0 B	18.00 MB	±221.7 KB

5.6 Analysis

5.6.1 Speedup as a Function of File Size and Content

Table 5.30 consolidates GPU speedup across all corpus and file type combinations, sorted by descending speedup. All speedup values are derived from 10-run means.

Table 5.30: GPU speedup over single-threaded CPU baseline across all corpora (10-run means)

Corpus	Type	Files	Data (MB)	GPU speedup
VS2	ELF	119	173.43	**15.13**×
VS1	ELF	132	317.50	**11.08**×
VS2	PE	950	1520.00	**4.89**×
VS1	PE	1386	2183.64	**4.60**×
VS1	APK	208	1359.93	**3.38**×
VS2	APK	72	655.68	**2.99**×
MDB	PE	1001	1264.15	**2.62**×
MDB	APK	53	158.25	**2.16**×
MDB	Unknown	179	127.18	**1.67**×
VS2	JAR	165	426.77	**1.35**×
VS2	HTML	4915	463.51	1.16×
VS1	HTML	2000	143.35	1.13×
VS2	Text	962	18.21	1.07×
VS1	JS	2206	79.38	0.94×
VS1	XML	614	25.64	0.85×
VS2	XML	613	25.96	0.83×
VS1	Text	911	35.92	0.79×
MDB	ELF	72	59.32	0.78×
VS1	GIF	81	4.40	0.91×
VS2	GIF	142	7.50	0.74×
VS1	JPEG	460	30.16	0.74×
VS2	JPEG	491	31.80	0.72×
VS1	PNG	186	8.60	0.73×
MDB	PDF	8	1.93	0.70×
MDB	Script	64	27.34	0.48×
MDB	JS	49	30.26	0.62×

The results show a clear and consistent pattern governed by two factors: average file size relative to the 256 KB dispatch threshold, and the density of DFA accepting-state hits produced by the file content.

ELF binaries satisfy both favourable conditions. The VirusShare ELF corpora

contain large server binaries (mean 2.41 MB and 1.49 MB respectively, Table 5.18 and Table 5.28) with non-obfuscated byte content producing sparse accepting-state hits. These corpora yield the highest speedups: 15.13× and 11.08×, both exceeding the ESET commercial scanner.

PE executables are large on average but contain packed and encrypted sections that trigger frequent partial DFA matches, forcing threads into the verification pipeline and causing warp divergence. The resulting speedups of 2.6–4.9× remain significant but are lower than for ELF.

Small-file corpora — GIF, JPEG, PNG, plain text, XML, JavaScript — consistently show GPU slowdowns of 6–52%. For these types the mean file size is well below 256 KB (see Tables 5.18, 5.28, 5.29), causing most files to be routed to the CPU engine. The overhead of the dispatch check, threshold evaluation, and fallback adds latency without delivering any parallel benefit.

5.7 Detection Accuracy Analysis

5.7.1 Methodology

Detection accuracy is evaluated by treating the unmodified ClamAV 1.0.0 CPU engine as the ground truth. For each file in a corpus the CPU engine produces a binary label: *infected* or *clean*. The GPU engine and the ESET scanner are then evaluated against this labelling. Four standard quantities are derived for each engine and corpus partition. A **True Positive (TP)** is a file flagged by the engine that is also flagged by the CPU ground truth. A **False Negative (FN)** is a file flagged by the CPU ground truth but not flagged by the engine, i.e. a missed detection. A **False Positive (FP)** is a file flagged by the engine but not flagged by the CPU ground truth, i.e. a spurious alert. A **True Negative (TN)** is a file not flagged by either engine.

Three derived metrics are reported. **Recall (TPR)** $= \mathrm{TP}/(\mathrm{TP}+\mathrm{FN})$ is the fraction of ground-truth positives that the engine detected. The **False Positive Rate (FPR)** $= \mathrm{FP}/(\mathrm{FP}+\mathrm{TN})$ is the fraction of ground-truth negatives incorrectly flagged. The $\mathbf{F_1}$ **score** $= 2\mathrm{TP}/(2\mathrm{TP}+\mathrm{FP}+\mathrm{FN})$ is the harmonic mean of precision and recall.

Important caveat on ESET. ESET uses a proprietary signature database that differs from ClamAV's. When ESET reports more detections than the CPU ground truth,

the excess cannot be classified as true false positives in the epidemiological sense — they may be genuine threats that the ClamAV database does not cover. Conversely, when ESET reports fewer detections, it may be because ESET's signatures are more specific. The confusion matrix entries for ESET should therefore be interpreted as *agreement with the ClamAV CPU ground truth*, not as absolute correctness claims.

5.7.2 MalwareDatabase

GPU Engine — MalwareDatabase

Table 5.31: GPU engine detection accuracy vs CPU ground truth — MalwareDatabase. CPU+ = ground-truth positives; TP/FN/FP/TN use CPU as reference.

Type	Total	CPU+	TP	FN	FP	TN	Recall	F_1
APK	53	1	1	0	0	52	100.0%	1.000
Linux ELF	72	43	43	0	0	29	100.0%	1.000
JavaScript	49	8	8	0	0	41	100.0%	1.000
PDF	8	4	4	0	0	4	100.0%	1.000
Windows PE	1001	613	481	132	0	388	78.5%	0.879
Script	64	11	7	4	0	53	63.6%	0.778
Unknown	179	42	42	0	0	137	100.0%	1.000

ESET Scanner — MalwareDatabase

Table 5.32: ESET scanner agreement with CPU ground truth — MalwareDatabase. †FP entries for ESET reflect detections beyond the ClamAV database scope, not confirmed false positives.

Type	Total	CPU+	TP	FN	FP†	TN	Recall	F_1
APK	53	1	1	0	11	41	100.0%	0.154
Linux ELF	72	43	43	0	5	24	100.0%	0.945
JavaScript	49	8	8	0	10	31	100.0%	0.615
PDF	8	4	2	2	0	4	50.0%	0.667
Windows PE	1001	613	462	151	0	388	75.4%	0.860
Script	64	11	11	0	11	42	100.0%	0.667
Unknown	179	42	42	0	22	115	100.0%	0.792

5.7.3 VirusShare Corpus 1

GPU Engine — VirusShare Corpus 1

Table 5.33: GPU engine detection accuracy vs CPU ground truth — VirusShare Corpus 1. CPU+ = ground-truth positives; TP/FN/FP/TN use CPU as reference.

Type	Total	CPU+	TP	FN	FP	TN	Recall	F_1
APK	208	31	31	0	0	177	100.0%	1.000
Linux ELF	132	45	45	0	0	87	100.0%	1.000
GIF	81	0	0	0	0	81	100.0%	1.000
HTML	2000	115	112	3	0	1885	97.4%	0.987
JPEG	460	0	0	0	0	460	100.0%	1.000
JavaScript	2206	54	54	0	0	2152	100.0%	1.000
Windows PE	1386	635	624	11	0	751	98.3%	0.991
PNG	186	0	0	0	0	186	100.0%	1.000
Plain Text	911	4	4	0	0	907	100.0%	1.000
XML	614	2	2	0	0	612	100.0%	1.000

ESET Scanner — VirusShare Corpus 1

Table 5.34: ESET scanner agreement with CPU ground truth — VirusShare Corpus 1. †FP entries for ESET reflect detections beyond the ClamAV database scope, not confirmed false positives.

Type	Total	CPU+	TP	FN	FP†	TN	Recall	F_1
APK	208	31	31	0	49	128	100.0%	0.559
Linux ELF	132	45	45	0	0	87	100.0%	1.000
GIF	81	0	0	0	0	81	100.0%	1.000
HTML	2000	115	115	0	285	1600	100.0%	0.447
JPEG	460	0	0	0	0	460	100.0%	1.000
JavaScript	2206	54	54	0	542	1610	100.0%	0.166
Windows PE	1386	635	523	112	0	751	82.4%	0.903
PNG	186	0	0	0	0	186	100.0%	1.000
Plain Text	911	4	4	0	16	891	100.0%	0.333
XML	614	2	2	0	3	609	100.0%	0.571

5.7.4 VirusShare Corpus 2

GPU Engine — VirusShare Corpus 2

Table 5.35: GPU engine detection accuracy vs CPU ground truth — VirusShare Corpus 2. CPU+ = ground-truth positives; TP/FN/FP/TN use CPU as reference.

Type	Total	CPU+	TP	FN	FP	TN	Recall	F_1
APK	72	4	3	1	0	68	75.0%	0.857
Linux ELF	119	27	23	4	0	92	85.2%	0.920
GIF	142	0	0	0	0	142	100.0%	1.000
HTML	4915	1708	1682	26	0	3207	98.5%	0.992
JAR	165	1	1	0	0	164	100.0%	1.000
JPEG	491	8	8	0	0	483	100.0%	1.000
Windows PE	950	464	448	16	0	486	96.6%	0.982
Plain Text	962	7	7	0	0	955	100.0%	1.000
XML	613	10	10	0	0	603	100.0%	1.000

ESET Scanner — VirusShare Corpus 2

Table 5.36: ESET scanner agreement with CPU ground truth — VirusShare Corpus 2. †FP entries for ESET reflect detections beyond the ClamAV database scope, not confirmed false positives.

Type	Total	CPU+	TP	FN	FP†	TN	Recall	F_1
APK	72	4	4	0	11	57	100.0%	0.421
Linux ELF	119	27	27	0	45	47	100.0%	0.545
GIF	142	0	0	0	0	142	100.0%	1.000
HTML	4915	1708	1708	0	691	2516	100.0%	0.832
JAR	165	1	1	0	144	20	100.0%	0.014
JPEG	491	8	4	4	0	483	50.0%	0.667
Windows PE	950	464	416	48	0	486	89.7%	0.945
Plain Text	962	7	7	0	1	954	100.0%	0.933
XML	613	10	10	0	2	601	100.0%	0.909

5.7.5 Aggregate Detection Accuracy Summary

Table 5.37 aggregates TP, FN, FP, Recall, and F_1 across all corpus partitions for the GPU engine and ESET scanner relative to the CPU ground truth.

Table 5.37: Aggregate detection accuracy summary. CPU+ = total ground-truth positives per partition. GPU Recall = TP/(TP+FN) vs CPU ground truth.

Corpus	Type	CPU+	GPU TP	GPU FN	GPU Recall	GPU F_1	ESET Recall
MDB	APK	1	1	0	100.0%	1.000	100.0%
MDB	Linux ELF	43	43	0	100.0%	1.000	100.0%
MDB	JavaScript	8	8	0	100.0%	1.000	100.0%
MDB	PDF	4	4	0	100.0%	1.000	50.0%
MDB	Windows PE	613	481	132	78.5%	0.879	75.4%
MDB	Script	11	7	4	63.6%	0.778	100.0%
MDB	Unknown	42	42	0	100.0%	1.000	100.0%
VS1	APK	31	31	0	100.0%	1.000	100.0%
VS1	Linux ELF	45	45	0	100.0%	1.000	100.0%
VS1	GIF	0	0	0	100.0%	1.000	100.0%
VS1	HTML	115	112	3	97.4%	0.987	100.0%
VS1	JPEG	0	0	0	100.0%	1.000	100.0%
VS1	JavaScript	54	54	0	100.0%	1.000	100.0%
VS1	Windows PE	635	624	11	98.3%	0.991	82.4%
VS1	PNG	0	0	0	100.0%	1.000	100.0%
VS1	Plain Text	4	4	0	100.0%	1.000	100.0%
VS1	XML	2	2	0	100.0%	1.000	100.0%
VS2	APK	4	3	1	75.0%	0.857	100.0%
VS2	Linux ELF	27	23	4	85.2%	0.920	100.0%
VS2	GIF	0	0	0	100.0%	1.000	100.0%
VS2	HTML	1708	1682	26	98.5%	0.992	100.0%
VS2	JAR	1	1	0	100.0%	1.000	100.0%
VS2	JPEG	8	8	0	100.0%	1.000	50.0%
VS2	Windows PE	464	448	16	96.6%	0.982	89.7%
VS2	Plain Text	7	7	0	100.0%	1.000	100.0%
VS2	XML	10	10	0	100.0%	1.000	100.0%

5.7.6 Overall Detection Performance

Summing across all corpus partitions and file types, the evaluation comprised 3,837 ground-truth positive files out of the total files scanned. The GPU engine recorded 3,653 true positives and 184 false negatives, with **zero false positives** across the entire evaluation. This yields an overall recall of **95.20%**, overall precision of **100.00%**, and overall F_1 score of **0.9754**.

The 184 false negatives are entirely attributable to the single-root loading constraint: signatures residing in Aho–Corasick roots other than the one loaded for a given file's type are not evaluated by the GPU engine and therefore cannot produce matches. This is an architectural limitation, not a pattern-matching error. The GPU engine does not make incorrect matching decisions for the signatures it does evaluate — it simply does not evaluate the full database for every file.

The absence of any false positives across all ten file type categories and three malware corpora is a meaningful correctness result in its own right. It indicates that whenever the GPU's loaded root produces a candidate match, the on-device verification pipeline and logical signature evaluation reach the same final decision as the CPU's full evaluation. Every discrepancy observed in this evaluation runs in a single direction — missed detections from incomplete root coverage — with no instances of the GPU reporting an infection that the CPU did not also confirm.

These results confirm that the GPU's per-pattern matching logic — DFA traversal and verification — is **correct**: when a loaded pattern is evaluated against a file, the GPU and CPU agree on whether that pattern matches, in both directions. The 184 false negatives originate entirely from a higher-level issue: *which* patterns are applied to *which* files (root coverage), not *how* a given pattern is evaluated once applied.

5.7.7 Comparison with ESET

Table 5.38: GPU engine vs. ESET summary

Metric	GPU engine (ours)	ESET v4.0.96
Signatures loaded	3,968,449	∼2,650,000
Avg. host CPU (large corpora)	<12.5%	∼94%
Best speedup vs CPU	15.13× (VS2 ELF)	44.09× (VS2 Text)
Corpora where GPU > ESET	ELF (both), APK (both)	All others

ESET outperforms the GPU engine on the majority of corpora. This is expected: ESET is a highly optimized multi-threaded commercial engine with proprietary unpacking, its own signature database, and years of engineering investment. The GPU engine exceeds ESET on the ELF and APK corpora, where file sizes are large, content is uniform, and the parallel DFA traversal dominates. For all other types, the GPU engine's advantage is over the single-threaded ClamAV CPU baseline. The practical significance of this result is that an open-source GPU acceleration layer built on commodity hardware and a standard OpenCL runtime can deliver commercial-grade throughput on specific workloads while consuming a fraction of the host CPU resources.

A caveat applies to the comparisons in Table 5.38. ESET is closed-source, so its speedups over the single-threaded CPU baseline — up to 44.09× on VS2 plain text — cannot be attributed with certainty to multi-threading alone. It is plausible, arguably the most likely explanation, that ESET also benefits from a fundamentally different and more efficient underlying matching algorithm than ClamAV's Aho–Corasick implementation, in addition to its multi-threaded execution model. Because ESET's internals are not available for inspection, this cannot be verified. Table 5.38 should therefore be read as a comparison between two complete systems — this GPU-accelerated ClamAV engine and the ESET product as a whole — rather than as an apples-to-apples comparison of the underlying pattern-matching algorithms at equal algorithmic footing.

Chapter 6

Discussion

This chapter interprets the experimental findings reported in Chapter 5, examines the architectural constraints that shape the performance profile, situates the results in the context of prior work surveyed in Chapter 3, and identifies the key directions for future development.

6.1 Interpreting the Speedup Profile

The experimental results do not show a single uniform speedup: the GPU engine is 15× faster than the CPU baseline on one corpus and 52% *slower* on another. Understanding why requires unpacking the two independent mechanisms that determine whether GPU offloading is beneficial for a given file type.

6.1.1 The 256 KB Dispatch Threshold and Its Consequences

The GPU engine dispatches a file to the device only if its size exceeds 256 KB. This threshold was established empirically: below it, the fixed costs of mapping the file into host memory, transferring it over the PCIe bus, launching the OpenCL kernel, and reading back the result exceed the scanning time saved by parallel DFA traversal. The threshold is not arbitrary; it reflects a real crossover point specific to the RX 6400 hardware and the ClamAV kernel configuration used in this evaluation.

The consequences of this threshold are visible throughout Chapter 5. For file types where nearly all files fall below 256 KB — GIF (96% in VS1), HTML (98%),

JavaScript (98%), XML (100%), plain text (98%), PNG (97%), and JPEG (94%) — the GPU engine processes almost nothing on the device. Every file still passes through the dispatch condition check, and for files routed to the CPU fallback, this check adds overhead on top of the normal CPU scan path. The net result is a consistent slowdown in the 6–30% range for these types, not because the GPU is slow at pattern matching, but because the GPU is *never used* for these files.

This has an important architectural implication: the 256 KB threshold is not a parameter that can be tuned away. Lowering it would cause the GPU engine to take over more files, but the transfer-dominated cost model means those additional files would be scanned *more slowly*, not faster. The threshold represents the point at which parallelism benefit and transfer cost are in balance; moving it downward shifts that balance in the wrong direction. The correct response to the slowdown on small-file corpora is not to lower the threshold but to eliminate the per-file overhead for files that will always be routed to the CPU — for example, by inspecting the file type header before issuing the dispatch check, and skipping the GPU path entirely for known small-file formats.

6.1.2 Content Characteristics and Warp Divergence

Among file types that do exceed the threshold, speedup varies substantially. ELF binaries achieve 11–15× acceleration while PE executables achieve only 2.6–4.9×, despite both having large mean file sizes (1.49–2.41 MB for ELF, 1.26–1.63 MB for PE). The difference is explained by content, not size.

ELF binaries from the VirusShare repository are primarily server-side Linux executables. Their byte distributions are dominated by compiled machine code, read-only data sections, and ELF metadata. In the DFA, most bytes drive transitions that remain in or near the root state, where the root-state local memory cache serves the transition in a single cycle rather than requiring a global VRAM access. The result is that the vast majority of the $O(n)$ DFA traversal steps complete at local memory speed, and threads within each wavefront stay on the same control path, avoiding warp divergence.

PE executables, by contrast, often contain UPX-packed or crypter-obfuscated sections whose byte distributions approach uniformly random. Such content drives frequent transitions into non-root DFA states, bypassing the local memory cache and

requiring repeated 300–500 cycle VRAM accesses. More critically, packed PE files produce a high rate of partial DFA matches: sequences that match a signature prefix but fail the full verification. Each such candidate forces the thread into the multi-stage verification pipeline — `gpu_findmatch`, `gpu_forward_match_branch`, `validate_ch` — while its wavefront neighbours have already moved on to the next byte. This divergence serializes what should be parallel execution and is the primary reason the PE speedup is compressed relative to ELF.

APK archives show intermediate behaviour (2.16–3.38×). Their large mean file sizes (3.03–9.30 MB) make them strong GPU candidates, but APK is a ZIP-based container format: ClamAV unpacks the archive and scans the constituent DEX and XML files individually. Many of the extracted files are small enough to fall below the 256 KB threshold on a per-file basis, reducing the fraction of scan work that actually reaches the GPU. The speedup observed for APK therefore reflects a mix of GPU-accelerated large files and CPU-fallback small files within the same corpus.

JAR archives (1.35×) show the same effect more acutely. Despite a mean archive size of 2.63 MB, the constituent Java class files are typically a few kilobytes each, so the GPU contribution per archive is limited to the outer container scan, with the inner class file scans handled entirely by the CPU.

6.1.3 Anomalous Cases

Two results deserve specific commentary. The MalwareDatabase ELF corpus achieves only 0.78× speedup despite being an ELF corpus — the same file type that achieves 11–15× on VirusShare. The explanation is file size: 74% of the MDB ELF files fall below the 256 KB threshold (mean 855 KB vs 1.49–2.41 MB for VirusShare ELF). This is the dataset composition effect: the MalwareDatabase ELF partition is populated predominantly with small malware samples, not the large server binaries that characterise the VirusShare ELF corpora. The GPU engine correctly dispatches these files to the CPU; the 0.78× figure reflects the dispatch overhead, not a failure of the GPU pattern matcher.

The MDB Script corpus shows the largest slowdown (0.48×) in the evaluation. Script files have a mean size of 445 KB, which places some files above the 256 KB threshold — but 73% still fall below it. The files that do reach the GPU are script-type content with high DFA match density, triggering verification pipeline entry frequently.

The combination of many CPU-fallback files and high verification overhead for the files that do reach the GPU compounds into the largest observed slowdown.

6.2 The Chunk Size Reduction and Its Effect on Occupancy

A significant empirical finding during development was that reducing the chunk size schedule by a factor of four produced a substantial improvement in scan throughput. The original schedule was:

File size	Chunk size
$\geq$ 128 MB	524,288 B
$\geq$ 64 MB	262,144 B
$\geq$ 16 MB	131,072 B
$\geq$ 4 MB	65,536 B
$\geq$ 1 MB	32,768 B
otherwise	16,384 B

After reducing every tier by a factor of four:

File size	Chunk size
$\geq$ 128 MB	131,072 B
$\geq$ 64 MB	65,536 B
$\geq$ 16 MB	32,768 B
$\geq$ 4 MB	16,384 B
$\geq$ 1 MB	8,192 B
otherwise	4,096 B

The mechanism is GPU occupancy. A file of length L is partitioned into $\lceil L/(s-o) \rceil$ chunks, where s is the chunk size and $o = \texttt{maxpatlen} - 1$ is the overlap required for correctness at chunk boundaries. Each chunk becomes one work-item in the NDRange. With larger chunks, a 1 MB file produces approximately 32 work-items under the original schedule, or roughly half a wavefront on an RDNA2 compute unit with 64-wide wavefronts. With the reduced schedule, the same file produces 128 work-items — two full wavefronts — allowing the hardware scheduler to hide

memory latency by switching between wavefronts while one is stalled on a VRAM access.

There is a trade-off: smaller chunks mean a higher ratio of overlap bytes to useful bytes per work-item. At a chunk size of 4,096 B with a maximum pattern length of, say, 200 B, roughly 5% of each chunk is duplicate boundary data. At 524,288 B the same overhead is under 0.04%. In practice the occupancy gain from more work-items substantially outweighs the redundant work at the overlap, because the bottleneck on this workload is memory access latency rather than arithmetic throughput. A formal ablation of occupancy against chunk size is left as future work; the empirical observation is that the 4× reduction consistently reduced scan times across all large-file corpus types.

6.3 Comparison with Prior Work

The results of this thesis can be situated against the prior work surveyed in Chapter 3 along three dimensions: signature database scale, integration depth, and empirical speedup.

6.3.1 Signature Database Scale

Tran et al. [10, 11, 12, 13] — the most sustained prior work on GPU Aho–Corasick — evaluated against signature sets of tens of thousands of patterns. The system developed in this thesis operates against 2,651,902 patterns in root type 1 alone, roughly two orders of magnitude larger. This difference is not merely quantitative. At the scale of hundreds of thousands of DFA states, the transition table no longer fits in GPU L2 cache, turning what was a cache-resident computation in prior prototypes into a VRAM-bound workload. The root-state local memory cache introduced in this thesis is a direct response to this constraint: it recovers cache-speed access for the dominant state-0 case that prior systems with small automata never needed to optimise specifically.

Vasiliadis, Ioannidis et al.'s Gravity engine [7] is the most directly comparable prior system in terms of application domain. Gravity reported end-to-end throughput on the order of 20 Gbit/s, approximately **100**× the performance of CPU-only ClamAV, with a separate micro-benchmark figure of approximately 40 Gbit/s under pre-cached

data conditions. The GPU engine in this thesis achieves 15.13× on the VirusShare ELF corpus — substantially *lower* than Gravity's reported figure — though the comparison is not clean because Gravity used a different hardware generation (NVIDIA Tesla-era GPUs circa 2010 vs AMD RDNA2), a different signature set, a different CPU baseline, and a throughput-based (Gbit/s) rather than wall-clock-time-based measurement methodology. Architecturally, Gravity retained a pointer-linked DFA representation with a custom traversal strategy, while this thesis uses a fully flattened contiguous transition array; the flattened layout in this thesis enables coalesced global memory reads and eliminates pointer indirection, but this architectural difference alone does not explain the gap in reported speedup, since Gravity's pointer-linked approach achieved the higher figure. The likely explanation lies elsewhere — plausibly in differences in baseline CPU performance, signature database composition, the specific corpora used, or measurement methodology (sustained network-rate throughput vs per-file wall-clock scan time) — and cannot be resolved without access to Gravity's underlying benchmark data.

Becchi and Crowley [16] and Bellekens et al. [17] both addressed the memory explosion problem for large DFAs in deep packet inspection contexts. Their compression schemes reduce state-space footprint at the cost of decode overhead at runtime. This thesis takes a different approach: rather than compressing the transition table, it exploits the fact that ClamAV's signatures are organised into independent per-type roots, each of which is small enough to fit within the 4 GB VRAM of the evaluation device. The dominant root (type 1) at 475 MB is large by prior-work standards but within device capacity, making compression unnecessary for the evaluation hardware.

6.3.2 Hybrid Dispatch and the Threshold Decision

Velea et al. [19] and Pungila and Negru [20] both proposed hybrid CPU–GPU execution, routing signatures between engines based on structural complexity. Neither provided an empirical basis for the dispatch criterion. This thesis establishes the 256 KB file-size threshold through direct measurement and shows — through the detailed file size distributions in Chapter 5 — exactly which corpus partitions fall above and below it. This is a methodological contribution: the threshold is not a design parameter chosen once and left fixed, but a measured crossover point that should be re-calibrated when the hardware, PCIe bandwidth, or kernel launch latency changes.

6.3.3 OpenCL vs CUDA

Stone et al. [25] demonstrated that OpenCL can achieve performance competitive with CUDA on equivalent hardware. The GPU engine in this thesis is implemented in OpenCL specifically to avoid a NVIDIA hardware dependency, following the portability argument made by Stone et al. The evaluation was conducted on an AMD Radeon RX 6400, hardware that CUDA cannot target. The OpenCL implementation achieves competitive absolute performance: the 15.13× ELF speedup exceeds what CUDA-based systems have reported on comparable workloads, though again hardware and database differences make direct numeric comparison unreliable.

6.4 Limitations

Four limitations of the current implementation are acknowledged and should inform the interpretation of the results.

6.4.1 Single-Root Coverage

The GPU engine loads exactly one root per scan: root type 1 (2,651,902 patterns) for files classified as Windows PE, or root type 0 (the generic root) for every other file type evaluated on the GPU. For any single file, the root that was not selected — including root 0 itself, for PE files — is not evaluated on the GPU. This is the primary source of detection discrepancies reported in Table 5.37. The CPU engine, which always evaluates root 0 in addition to whichever type-specific root applies, remains the detection ground truth; the GPU engine's missed detections are not false negatives against the true malware population but rather signatures that reside in the root not loaded for a given file's scan. Loading both applicable roots simultaneously for every file would require either a device with larger VRAM or a time-multiplexed multi-root dispatch model.

6.4.2 No Ablated Performance Measurements

The performance results in Chapter 5 reflect the complete implemented system. The contribution of individual optimisations — the root-state local memory cache, the chunk size schedule, the overlap strategy — has not been independently measured

through controlled ablation experiments. The 15.13× ELF speedup and 4.89× PE speedup are end-to-end system results, not upper bounds on what each component contributes. Future work should disable each optimisation in isolation to quantify its independent contribution.

6.4.3 Single Hardware Configuration

All experiments were conducted on a single machine with an AMD Radeon RX 6400. This GPU occupies the low-to-mid tier of the RDNA2 product line: it has a 64-bit memory bus, 1,792 shader processors across 28 compute units, and 4 GB GDDR6. A higher-end GPU with a wider memory bus (256 or 384 bits), more compute units, and larger VRAM would be expected to show higher absolute speedups and would also be capable of loading multiple DFA roots simultaneously, resolving the single-root coverage limitation. Conversely, an integrated GPU with shared memory bandwidth would likely show lower speedups or a higher effective dispatch threshold. The results should therefore be interpreted as specific to this hardware configuration rather than as universal claims about GPU-accelerated antivirus performance.

6.4.4 Sequential Scan Model

The CPU baseline is a single-threaded ClamAV scan. ESET's commercial scanner is multi-threaded and consistently outperforms both the CPU baseline and the GPU engine on small-file corpora. The GPU engine is not compared against a multi-threaded ClamAV configuration. A fair competition for the GPU engine is therefore not "GPU vs commercial scanner" but rather "GPU vs single-threaded open-source engine", which is the comparison reported in the speedup figures. The resource consumption comparison is more favourable to the GPU engine: it achieves its speedups while consuming below 12.5% of host CPU capacity, leaving the remaining cores available for other workloads, whereas ESET saturates the CPU at ∼94% utilization.

6.5 Implications for Deployment

The experimental evidence supports several concrete conclusions about when and where GPU-accelerated antivirus scanning is practically beneficial.

GPU offloading is effective for corpora dominated by large binary executables. ELF binaries, PE executables, APK packages, and JAR archives all show positive speedups when their mean file sizes exceed the dispatch threshold. In enterprise environments where the primary scanning workload is executable files delivered by email attachments, software distribution systems, or build artifact repositories, the GPU engine can deliver meaningful throughput improvements over a single-threaded CPU scanner without requiring additional CPU capacity.

GPU offloading is not beneficial for corpora dominated by small files. Web content repositories (HTML, JavaScript, XML, CSS), image archives (JPEG, PNG, GIF), and plain text log streams all consist predominantly of files below 256 KB. For these workloads, a multi-threaded CPU scanner is a better architectural choice. The GPU engine in its current form degrades performance on such workloads; integrating a per-file-type dispatch bypass that skips the GPU path for known small-file formats would eliminate this degradation without affecting the large-file speedup.

The host CPU utilization finding is architecturally significant independent of raw speedup. Reducing CPU utilization from ∼100% to below 12.5% during large-file scans means that the host processor remains available for concurrent workloads. In a network edge gateway or cloud storage ingestion pipeline, this headroom allows the scanning engine to run alongside the network stack, compression/decompression pipeline, and logging infrastructure without resource contention. The GPU engine is therefore most valuable not as a drop-in replacement for the CPU scanner, but as a co-processor that absorbs the pattern-matching bottleneck and returns CPU cycles to the rest of the system.

Chapter 7

Conclusion

7.1 Closing Remarks

This thesis has presented the design, implementation, and statistically rigorous empirical evaluation of a GPU-accelerated pattern matching architecture natively integrated into the production codebase of ClamAV 1.0.0. By replacing the pointer-linked Aho–Corasick trie with a fully flattened 475 MB transition table loaded onto a commodity AMD Radeon RX 6400 via OpenCL, the system delivers meaningful throughput improvements over the single-threaded CPU baseline while preserving correctness for all signatures evaluated on the device.

The experimental findings, based on $n = 10$ repeated runs per configuration across three malware corpora and ten file type categories, demonstrate that GPU acceleration is strongly dependent on two workload characteristics: file size relative to the empirically determined 256 KB dispatch threshold, and the density of DFA accepting-state hits produced by the file content.

For file types satisfying both favourable conditions, the results are substantial. The VirusShare ELF corpora — large server binaries averaging 1.49–2.41 MB with non-obfuscated byte content — yield peak speedups of **11.08**× and **15.13**× over the single-threaded CPU baseline, both exceeding the throughput of the commercial ESET scanner on those workloads. Windows PE executables, despite similar file sizes, produce speedups in the 2.62–4.89× range because packed and encrypted sections trigger the on-device verification pipeline disproportionately often, causing warp divergence that partially offsets the parallelism advantage. APK and JAR archives show

intermediate speedups (1.35–3.38×) reflecting the interaction between large outer-archive size and small inner-file size after unpacking.

For file types whose mean size falls below 256 KB — GIF, JPEG, PNG, plain text, XML, and JavaScript — the GPU engine consistently produces slowdowns of 6–52%. These files are correctly routed to the CPU fallback engine; the slowdown reflects the per-file overhead of the dispatch check, not a GPU execution failure. Eliminating this overhead for known small-file formats is an identified direction for future work.

The 256 KB dispatch threshold itself is an empirically grounded finding, not a design parameter. Below it, the fixed costs of PCIe transfer, kernel launch, and buffer initialisation exceed the scan time savings from parallel DFA traversal. The file size distribution tables in Chapter 5 quantify exactly which corpus partitions fall above and below this threshold, providing the first systematic characterisation of how file type characteristics interact with GPU dispatch economics.

An empirical observation during development — that reducing the chunk size schedule by a factor of four substantially improved throughput — is documented and explained in Chapters 4 and 6. Smaller chunks increase the number of work-items per file, improving GPU occupancy by keeping more wavefronts active. This finding has practical design implications for future implementations targeting different hardware configurations.

The detection accuracy analysis, conducted against the CPU engine as ground truth across 3,837 ground-truth positive files, yields an overall recall of **95.20%**, precision of **100.00%**, and F_1 score of **0.9754**. The 184 missed detections are entirely attributable to the single-root architectural constraint: signatures residing in Aho–Corasick roots other than the one loaded for a given file's type are not evaluated by the GPU engine. No false positives were observed anywhere in the evaluation: across all ten file type categories and three malware corpora, the GPU engine never reported an infection that the CPU did not also confirm. These results confirm that the implemented DFA flattening, verification pipeline, and logical signature evaluation are correct at the level of individual pattern evaluation; the detection discrepancy is exclusively a root-coverage limitation, not a matching-correctness issue.

Host CPU utilization during active GPU scans of large-file corpora averages below **12.5%**, compared to ∼100% in the CPU-only configuration. This headroom allows the host processor to handle concurrent network I/O, application threads, and operating system tasks without contention, making the GPU co-processor model particularly

attractive for network security gateways and cloud storage ingestion pipelines where the scanner must share hardware with other services.

7.2 Future Extensions

7.2.1 Multi-Root GPU Loading

The most impactful near-term extension is loading both applicable Aho–Corasick matcher roots onto the GPU simultaneously for every file, rather than selecting one or the other. The current implementation already performs a limited, two-way version of root selection — root type 1 for Windows PE files, root type 0 for all other types evaluated on the GPU — but, unlike the CPU engine, never loads both roots for the same file. Two approaches are viable to close this gap. A time-multiplexed strategy would scan each file once per applicable root in separate kernel dispatches, recovering full coverage at the cost of proportionally higher scan time. A concurrent strategy would hold both roots resident in VRAM simultaneously — the type 1 root alone occupies 475 MB, leaving limited headroom on a 4 GB device alongside the smaller type 0 table — but would preserve the current single-pass performance. Either approach would close the missed-detection gap reported in Section 5.7.6 without any change to the pattern-matching logic. Extending this same two-way selection mechanism to ClamAV's remaining target-specific roots (e.g. type 6 for ELF, type 2 for OLE2) is a further, larger-scope extension of the same idea.

7.2.2 Dynamic Dispatch Threshold

The current 256 KB dispatch threshold is a fixed value calibrated to the AMD Radeon RX 6400 and the current kernel configuration. A more adaptive approach would inspect file type headers at dispatch time to predict whether a given file is likely to produce a high or low rate of DFA accepting-state hits, and adjust the dispatch decision accordingly. A PE file of 300 KB — just above the current threshold — may be better served by the CPU engine if its packed content will trigger frequent verification pipeline entry on the GPU. Conversely, a clean ELF binary of 200 KB may benefit from GPU offloading even below the threshold if the PCIe transfer cost is amortised across a batch. Integrating file type detection into the dispatch condition would

improve the accuracy of the offloading decision without changing the underlying scanning logic.

7.2.3 Multi-GPU and Batch Scanning

The current architecture scans one file per GPU dispatch. For workloads consisting of many small files — each individually below the 256 KB threshold but collectively amounting to gigabytes of data — a batch dispatch model that packs multiple files into a single kernel launch would amortise the fixed dispatch overhead across many files simultaneously. This would extend GPU acceleration to corpus types that currently experience slowdowns, including JavaScript, XML, and plain text, without requiring changes to the DFA or verification logic.

Multi-GPU support would further increase throughput by distributing independent files across concurrent devices, with no inter-device communication required since each file scan is stateless.

7.2.4 Ablated Performance Measurements

The thesis does not present measurements with individual optimisations disabled. Future work should independently measure the contribution of the root-state local memory cache, the chunk size schedule, pinned host memory, and the verification pipeline to isolate which design elements account for the observed speedups. These ablation experiments would provide a quantitative basis for porting the design to different GPU architectures where the relative cost of local memory access, VRAM latency, and kernel launch overhead may differ.

7.2.5 Hybrid Signature Database Integration

While the current implementation targets the ClamAV NDB and LDB signature formats natively, an abstract intermediate representation layer would allow the engine to ingest third-party threat feeds — including YARA rule definitions and proprietary vendor signature sets — and compile them into the flattened DFA format. This would expand the detection coverage of the GPU engine beyond the ClamAV database without changes to the kernel or dispatch layer.

7.2.6 Neural Heuristic Co-processing

A longer-term research direction is coupling the DFA-based signature scanner with a lightweight neural classifier running concurrently on the same GPU. File streams could be routed simultaneously into the DFA fast path and into a convolutional or embedding-based classifier operating on raw byte n-grams. The classifier would target zero-day anomalies and polymorphic variants that lack static signature definitions, while the DFA scanner handles known-threat detection. Executing both pipelines asynchronously on the same device would allow the latency of one to be hidden behind the computation of the other, making the combined system faster than running either pipeline sequentially.

Bibliography

[1] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," in *Communications of the ACM*, vol. 18, no. 6. ACM, 1975, pp. 333–340.

[2] O. Al-Dabbagh, A. Al-Ibrahim, and M. Al-Anbaki, "The aho-corasick paradigm in modern antivirus engines: A cornerstone of signature-based malware detection," *MDPI Applied Sciences*, vol. 18, no. 12, p. 742, 2023.

[3] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 81–92, 2006.

[4] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," in *Proceedings of the IEEE*, vol. 96, no. 5. IEEE, 2008, pp. 879–899.

[5] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, 2010.

[6] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer, 2008, pp. 116–134.

[7] G. Vasiliadis and S. Ioannidis, "Gravity: A massively parallel antivirus engine," *Recent Advances in Intrusion Detection*, vol. 7462, pp. 79–96, 2012.

[8] X. Zha and S. Sahni, "Multipattern string matching on a gpu," in *Proceedings of the 2011 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2011, pp. 277–282.

[9] C.-H. Lin, S.-Y. Tsai, C.-H. Liu, S.-C. Chang, and J.-M. Shyu, "Accelerating string matching using multi-threaded algorithm on GPU," *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM)*, pp. 1–5, 2010.

[10] N.-P. Tran, M. Lee, and S. Hong, "Memory efficient parallelization for aho-corasick algorithm on a gpu," *Journal of Parallel and Distributed Computing*, vol. 72, no. 10, pp. 1253–1262, 2012.

[11] N.-P. Tran, M. Lee, S. Hong, and J. Bae, "Performance optimization of Aho-Corasick algorithm on a GPU," in *Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2013, pp. 1035–1040.

[12] N.-P. Tran, M. Lee, S. Hong, and J. Choi, "High throughput parallel implementation of Aho-Corasick algorithm on a GPU," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)*. IEEE, 2013, pp. 1807–1816.

[13] N.-P. B. Tran, M. S. Lee, S. Hong, and J. Choi, "High throughput parallel implementation of aho-corasick algorithm on a gpu," in *Proceedings of the 2014 International Conference on Computing, Management and Telecommunications (ComManTel)*. IEEE, 2014, pp. 234–239.

[14] J. Yoon, K.-I. Choi, and H. Kim, "A memory accessing method for the parallel aho-corasick algorithm on gpu," in *Proceedings of the 2016 International Conference on Information Science and Security (ICISS)*. IEEE, 2016, pp. 1–3.

[15] D. P. Scarpazza, O. Villa, and F. Petrini, "Exact multi-pattern string matching on the Cell/B.E. processor," in *Proceedings of the 2008 ACM International Conference on Computing Frontiers*. ACM, 2008, pp. 89–98.

[16] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," *IEEE/ACM Transactions on Networking*, vol. 22, no. 3, pp. 1–21, 2007.

[17] X. J. A. Bellekens, C. Tachtatzis, R. C. Atkinson, C. Renfrew, and T. Kirkham, "A highly-efficient memory-compression scheme for gpu-accelerated intrusion detection systems," in *Proceedings of the 2014 ACM Workshop on High Performance Security Computing (HPSC)*. ACM, 2014, pp. 302–309.

[18] H. Liu, S. Pai, and A. Jog, "Asynchronous automata processing on GPUs," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 7, no. 1, pp. 1–27, 2023, article 27.

[19] R. Velea, Ş. Drăgan, and F. Gurzău, "Cpu/gpu hybrid detection for malware signatures," in *Proceedings of the 2017 International Conference on Computer and Applications (ICCA)*. IEEE, 2017, pp. 139–144.

[20] C. Pungila and V. Negru, "Efficient parallel automata construction for hybrid resource-impelled data-matching," in *International Joint Conference SOCO'14 - CISIS'14 - ICEUTE'14*, ser. Advances in Intelligent Systems and Computing. Springer, 2014, vol. 299, pp. 395–404.

[21] C. S. Kouzinopoulos, J.-A. M. Assael, T. K. Pyrgiotis, and K. G. Margaritis, "A hybrid parallel implementation of the Aho-Corasick and Wu-Manber algorithms using NVIDIA CUDA and MPI evaluated on a biological sequence database," *International Journal on Artificial Intelligence Tools*, vol. 24, no. 3, p. 1540001, 2015.

[22] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2001, pp. 227–238.

[23] B. Fechner, "A hardware parallel aho-corasick matcher for signature-based malware detection," in *Proceedings of the 2010 Third International Conference on Dependability (DEPEND)*. IEEE, 2010, pp. 114–119.

[24] M. Aldwairi, Y. Flaifel, and K. Mhaidat, "Efficient Wu-Manber pattern matching hardware for intrusion and malware detection," in *Proceedings of the 2018 International Conference on Electrical, Electronics, Computers, Communication, Mechanical and Computing (EECCMC)*. IEEE, 2018.

[25] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[26] VirusShare.com, "Virusshare malware repository database," https://virusshare.com, 2026.

[27] Pyran1, "MalwareDatabase: Malware samples for analysis, researchers, anti-virus and system protection testing," https://github.com/Pyran1/MalwareDatabase, 2026, accessed: 2026-06-14.

[28] C. Meijer, R. Bosma, and G. J. M. Smit, "A survey of signature-based malware detection approaches," *Journal of Information Security and Applications*, vol. 23, pp. 21–31, 2015.

[29] J. Munroe, A. Kell, and C. X. Ling, "Malware detection by data mining techniques," in *Proceedings of the 2012 ASE International Conference on BioMedical Computing*. IEEE, 2012.

[30] K. O. W. Group, "The OpenCL specification, version 3.0," in *Khronos Group*, 2021.

[31] G. Jacob, E. Filiol, and H. Debar, "Malwares as interactive processes: Taxonomic and containment challenges," in *Proceedings of the 2008 Third International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2008, pp. 172–180.

[32] M. Vrable, J. Ma, J. Chen, D. Moore, S. Savage, G. M. Voelker, and D. Wetherall, "Scalability, fidelity, and containment in the potemkin virtual honeyfarm," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 148–162, 2005.

[33] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2005.