Reconfigurable Computing Challenge: RapidScan High-Throughput Parameterized HLS-based Streaming String Matching Library for FPGAs

Shashank Obla*, Tommy Tracy II†, Matthew Beck†, James C. Hoe*, Kevin Skadron†, Wajih Ul Hassan†
*Carnegie Mellon University, Pittsburgh, PA, USA †University of Virginia, Charlottesville, VA, USA
sobla@andrew.cmu.edu, tjt7a@virginia.edu

***Abstract*—Prior work in hardware acceleration of regular-expression matching deep-packet inspection has shown orders of magnitude improvement in energy and cost. However, efficiency gains from an RTL implementation lack portability across domains, FPGA platforms and even rulesets or input traffic patterns that demand re-tuning of the streaming input-dependent filtering stages. In this work, we introduce RapidScan, a novel HLS-based streaming string-matching library that treats parameterizability and development agility as first-class design goals alongside performance and efficiency.**

Using the streaming computation model, we designed the data-dependent computations of string matching into a library of fully-pipelined HLS kernels. The kernels can be composed into a streaming pipeline at compile time as well as parameterized to tradeoff performance and resources for specific workloads. To demonstrate the library's efficacy, we showcase its application to log monitoring on the Versal V80 device, adding new kernels essential for processing logs. The resulting system, RapidDetect, achieves a throughput of over 160 Gbps on real-world log inputs using a single server.

I. Introduction

As hyperscalers push the limits of data centers, the scalability of log monitoring has become critical for delivering efficient security solutions. However, existing log-based systems remain bottlenecked by batch-oriented architectures that require database ingestion prior to scheduled rule evaluation. Streaming implementations using software-based regular-expression libraries such as Hyperscan [1] can reduce detection latency and provide orders-of-magnitude higher bandwidth than database query-based solutions. Nevertheless, the fundamental operation underlying regex matching, string matching, is ill-suited for general-purpose hardware, requiring hundreds of CPU cores to saturate network line rates of 200 Gbps and beyond.

Pigasus [2], a regex engine designed for network security, employs FPGA-based pre-filtering stages operating at line rates of 100 Gbps to rapidly discard traffic that fails to meet baseline rule conditions. This reduces the dependency on expensive regular expression matching engines in software, providing an extremely fast and energy- and cost-efficient solution on a single server. However, this common-case optimization strategy renders the RTL solution fragile to changes in input traffic composition and detection rulesets. Moreover, porting the design to the new domain of log monitoring introduces further inefficiencies that cannot be solved purely by retuning the existing design.

This work was supported in part by PRISM, one of seven centers in JUMP 2.0, an SRC program sponsored by DARPA, and National Science Foundation (NSF) under Award No. 2339483. We would also like to acknowledge AMD for enabling the work through the Heterogeneous Accelerated Compute Clusters (HACC) Program. RapidScan is part of the larger RapidDetect project, to develop open-sourced FPGA-accelerated text and data analytics for real-time event and log monitoring capabilities.

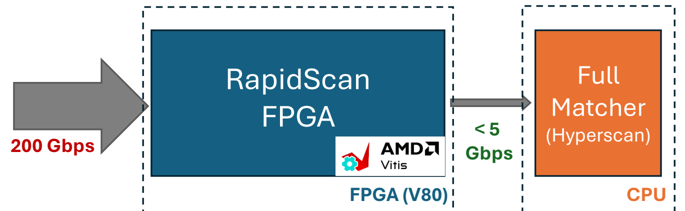

Fig. 1: RapidScan String Matching operates at over 200Gbps using less than 30% of the Versal V80. Along with Hyperscan on the host CPU, the system, RapidDetect, can process incoming logs at over 160Gbps on a single server

To provide a parameterizable library of string matching filters that can be rapidly adapted to new domains and workloads, we developed a streaming FPGA string matching library: RapidScan based on the kernels from Pigasus. Written in High-Level Synthesis (HLS), RapidScan exposes a broad design space of throughput and resource tradeoffs for individual string matching filters. In addition, the ability to compose existing library components with new domain-specific kernels allows RapidScan to provide high-throughput FPGA-based string matching solutions across workloads and domains, while minimizing energy and resource costs.

We use RapidScan to develop RapidDetect[1], a high-performance streaming log monitoring pipeline (shown in Figure 1). To achieve this, we designed novel kernels specifically for parsing JSON-formatted logs and enhanced existing filters to robustly process Sigma rules [3]. The prototype, implemented on a AMD Versal V80 FPGA, utilizes the on-board High-Bandwidth Memory (HBM) to feed the RapidDetect pipeline and employs QDMA for communication with Hyperscan on the host CPU. Our solution achieves over 160 Gbps of throughput on real-world log traces, evaluating over 4,000 rules and approximately 18,000 string literals, delivering a 30x speedup compared to a CPU-only Hyperscan baseline.

[1] Open-sourced at https://github.com/pigasus-hls/RapidDetect-RCC

II. Background

A. Input-Dependence and Common-Case Optimization

Traditional hardware accelerators target computations with regular parallelism such as matrix multiplication or convolution since they statically map to spatial hardware structures effectively. Parallelism in input-dependent applications, such as string matching, on the other hand, is irregular and varies with the content of the data being processed. For example, an input stream that matches a larger number of strings or rules will produce more work for downstream stages. Without any knowledge of the input traffic or ability to change the design once deployed, the accelerator would have to be designed assuming the worst-case input.

[4] argues for common-case optimization for input-dependent workloads when designing for the worst-case can be prohibitively expensive in hardware. Common-case optimization has two main implications on streaming accelerator design: (1) The design must be parameterizable and adaptable to allow for retuning to changing common-cases or deployment conditions, and (2) Infrastructure kernels must be introduced that handle the data-dependent design patterns such as compaction to densify sparse intermediate data streams.

B. Pigasus Intrusion Prevention System

Pigasus [2] is an FPGA-first regular-expression matching engine for network intrusion detection/prevention. Using common-case optimization strategies, Pigasus implements hierarchical filtering stages on the FPGA ending with full regular-expression matching on the host CPU. With each successive filter, although the compute complexity rises, the throughput required drops precipitously as traffic and the set of potential rule matches get filtered in the common-case.

The two string matching filters in Pigasus are the (1) Multi-String Pattern Matcher (MSPM) and the (2) Conjunct Pattern Matcher (CPM). The MSPM scans packet payloads at line rate for many thousands of string literals, extracted from the regex-based rules, concurrently. This fast and cheap filtering using hash-table and shift-or-based scanning removes a large fraction of incoming traffic. The CPM performs more selective checks but only for rules that match in the MSPM, significantly reducing the complexity and speed of scanning needed. Using a bloom-filter-like fingerprint, the CPM checks for conjunctions of literals, significantly reducing false positives.

C. High-Level Synthesis

Commercial High-Level Synthesis (HLS) [5], [6] tools have improved significantly over the past decade in making hardware design more accessible [7]. HLS allows hardware to be expressed in a high-level language such as C++, using most of the language features for compile-time expressability such as object-oriented features and template meta-programming, essential to designing parameterizable hardware designs.

Many research works have explored automatic reasoning for HLS optimization and design-space exploration (DSE) (such as [8]). However, the solutions for input-dependent irregular computations remain suboptimal where the onus for system design and kernel de-composition still lies with the developer. *A kernel with data-dependence embedded in its functionality cannot be optimized with simple transformations within the kernel itself.*

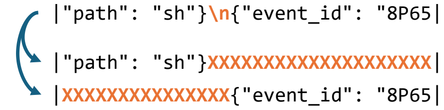

Fig. 2: The input log stream must be split into packets using the newline character as the delimiter. Padding bytes are inserted to fill to the stream width.

III. Input-Dependent Streaming Pipelines in HLS

As discussed in Section II-C, representing irregular parallelism found in input-dependent computations can be challenging in High-Level Synthesis (HLS). Simple data-dependencies such as variable loop-bounds often prevent the HLS compiler from applying certain optimizations. To wrangle the complexity in a large input-dependent pipeline, we use the Streaming Data paradigm to decompose the pipeline into smaller, more manageable kernels. Each kernel performs spatially parallel regular computations amenable to HLS optimization, while the input-dependence is decoupled across kernels using stream backpressure. The computation in each kernel follows the structure shown below:

```
template <typename T>
void kernel(hls::stream<T> in, hls::stream<T> out) {
  "architectural state" variables;
  while (true) {
    "transient state" variables;

    // could read multiple pipes
    T in_flit = in.read();

    // loop body (can do anything that statically elaborates
    ↪  into a dataflow)

    // could write multiple pipes
    out.write(out_flit);
  }
}
```

As long as care is taken to ensure that the dependencies across loop iterations expressed through the "architectural state" variables can be computed within one cycle, the loop can be fully pipelined to achieve maximum throughput.

A. Compute Decomposition Example

Consider a source module that must split the input stream flits based on the newline character to produce log event packets for further processing as shown in Figure 2. This computation, if represented using a single kernel, prevents the HLS compiler from fully pipelining the loop since depending on the input data, a single flit might produce either one or two output flits, which cannot be computed within a single cycle.

This issue is remediated by splitting the computation into two kernels connected by two intermediate streams, one for regular flits and the other for when a newline is detected. The second kernel can, based on metadata carried by the flits, easily interleave the flits from the input streams while producing the necessary backpressure to throttle the upstream kernel since the total throughput is limited to the width of a single stream.

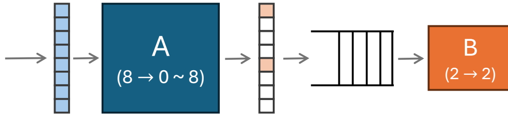

Fig. 3: Kernel A produces 2 elements per flit on average which can be handled by a narrower kernel B. The sparse intermediates must be compacted to create a denser flit for kernel B to process efficiently.

B. Compaction Kernel

Input dependence can also manifest as more complex patterns, such as sparsity within the flits of a data stream, which necessitates a dedicated compaction kernel. Consider the system shown in Figure 3. A streaming module A consumes 8 elements of the input stream in parallel every cycle and, depending on the input values, produces between 0 and 8 output elements (averaging 2 elements). To efficiently drive a narrower downstream kernel B, this sparse intermediate stream must be densely packed, allowing kernel B to be unrolled without requiring additional logic to skip over empty entries.

While a compaction kernel can be easily written in HDL as a tree of 2-to-1 stages separated by buffers, creating a fully parameterized C++ implementation of this inherently spatial structure is non-trivial. To mimic the hardware reduction tree, we opt for a similar approach by specializing a 2-to-1 stage and use template recursion to, at compile time, instantiate the tree with a customizable compaction factor.

```
template <int FanIn>
struct compactor_t<FanIn, 1> {
  template <typename T>
  static void compactor(stream<T> *in_pipes,
                        stream<T> out_pipe) {
    stream<T> inter_pipe_0, inter_pipe_1;
    task core_task_0(compactor_t<FanIn / 2, 1>::template
    ↪  compactor<TPayload>,
                     &in_pipes, &inter_pipe_0);
    task core_task_1(compactor_t<FanIn / 2, 1>::template
    ↪  compactor<TPayload>,
                     &in_pipes[FanIn / 2], &inter_pipe_1);
    task merge_task(compact_2to1<TPayload>, inter_pipes_0,
    ↪  inter_pipes_1, out_pipe);
  }
};
```

C. Parameterization

HLS allows us to leverage many of the advanced language features inherent to C++. We implement parameterization using compile-time macros to control the unrolling of each kernel, enabling the tradeoff between throughput and resource utilization based on the workload's common case. The functionality within the kernels can also be altered to tailor the kernel to the needs of a specific domain. Furthermore, templates parameterize stream types and kernel functions, facilitating the type-checked composition of the end-to-end pipeline.

IV. RapidScan Kernels

The primary compute kernels in RapidScan comprise the MSPM and CPM, which are derived from Pigasus [2] (Figures 4 and 5). We completely redesigned these kernels using the HLS design pattern detailed in Section III, incorporating extensive functional and performance parameterization. Furthermore, we implemented specialized infrastructure kernels to

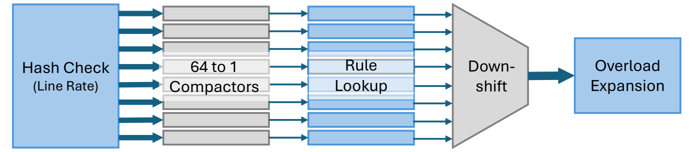

Fig. 4: Multi-String Pattern Matcher Pipeline

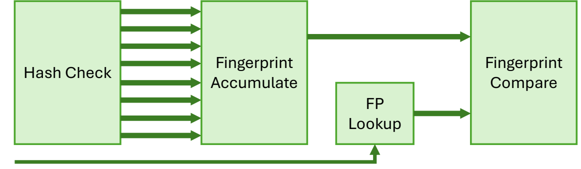

Fig. 5: Conjunct Pattern Matcher Pipeline

seamlessly interconnect the main compute modules, handling necessary data-width conversions, and compaction both within the MSPM (for efficient rule lookups) and across the filters.

A. Kernels for Log Monitoring

Log Monitoring, being a different domain from network security, has unique characteristics that the original Pigasus MSPM and CPM functionalities could not effectively accommodate. Specifically, the logs and their associated rules are JSON-formatted, consisting of `"field": "value"` pairs with a small set of unique field types that could not be natively represented in the existing kernels. Furthermore, most Sigma rules [3] are of the Conjunctive Normal Form (ANDs of ORs). Because the CPM can only process strict conjunctions, these rules must be decomposed into individual conjunction rules that often share multiple literals, creating conflicts that must be resolved into all possible matches before further processing.

To address these challenges, we introduced two entirely new stages: (1) a Field Tagger and (2) an Overload Expansion module within the MSPM. The Field Tagger parses JSON-formatted logs for predetermined field literals and appends metadata to the logs, marking the position of bytes belonging to the particular field's value. As the first stage in the pipeline, it must sustain line-rate traffic and, therefore, is implemented as a fully unrolled, deeply pipelined HLS kernel. We modified the MSPM and CPM kernels to utilize this field-tagged metadata to perform field-aware scanning of the log stream, improving the selectivity for Sigma rules. Overload Expansion is a new stage integrated into the MSPM pipeline to resolve conflicts caused by multiple rules sharing the same fast pattern. By employing a fast-slow path common-case design pattern [4], it maintains high throughput for non-conflicting rules while processing conflicted rules without stalling.

V. RapidDetect on Versal V80

Using the kernels in RapidScan, we implemented a fully functional prototype on the Versal V80 FPGA, utilizing AVED as a starting point (Figure 6). QDMA facilitates data transfer to and from the FPGA memories, which includes sending the filtered log stream back to the host for Hyperscan processing. Hyperscan runs as a separate, multi-threaded process that

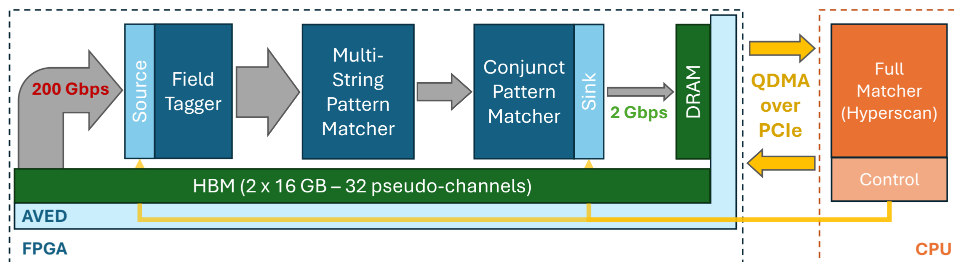

Fig. 6: RAPIDDETECT System Prototype implemented on the Versal V80

TABLE I: Resource Utilization of the kernels and the total RAPIDDETECT Pipeline on the Versal V80 Accelerator

Module	LUTs	DSP	FF	BRAM	URAM
RAPIDDETECT	175,964	0	279,482	938	13
Field Tagger	5,263	0	8,475	0	0
MSPM	123,831	0	185,885	576	0
CPM	25,063	0	33,966	271	0

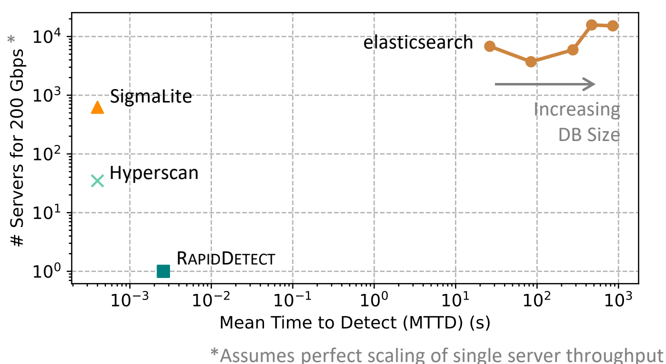

Fig. 7: Number of servers needed to scale to 200 Gbps vs Mean Time to Detect for different Log Monitoring solutions. RAPIDDETECT achieves near 200Gbps throughput on a single FPGA-enabled server.

communicates with the FPGA host code via shared memory. Table I details the system's resource utilization. We utilized the HBMs as a stand-in for future Ethernet integration. To harness their full capacity and support longer traces, we configured the multiphase NoC mode to guarantee full-bandwidth QoS to all HBM ports from the design's source kernels. Finally, we floorplanned RAPIDDETECT to the HBM-adjacent SLR, restricting SLR crossings to only NoC-routed control signals.

VI. EVALUATION

We evaluate RAPIDDETECT using Linux Sigma rules [3] over log traces from DARPA's Transparent Computing (TC) program [9]. An Intel Core i9-13900K CPU is used to evaluate the software baselines and serves as the host for the RAPIDDETECT pipeline. Figure 7 plots the number of servers required to sustain 200 Gbps of traffic against threat detection latency. RAPIDDETECT achieves 200 Gbps on a single server, demonstrating orders of magnitude lower latency than the state-of-the-art commercial solution, Elasticsearch [10]. While Hyperscan achieves a lower Mean Time to Detect (MTTD), scaling it to 200 Gbps would require approximately 40 servers. Consequently, even though the V80 FPGA costs ten times as much as the server, RAPIDDETECT remains over 4x cheaper than an ideally scaled Hyperscan. Elasticsearch, limited by its sub-1 Gbps throughput, would require over 2,000 servers, making it highly cost-prohibitive to scale to 200 Gbps.

VII. CONCLUSION

We introduced RAPIDSCAN, an HLS-based library of high-throughput parameterizable and composable input-dependent string matching kernels. Leveraging this library, we developed RAPIDDETECT, a system that achieves near 200 Gbps throughput for Log Monitoring at one-fourth the cost of a comparable software-only solution. The expansive fabric and HBM capacity of the V80, coupled with the adaptability of RAPIDSCAN, will allow us to extend RAPIDDETECT to address more complex challenges, such as APT detection using provenance graph analysis [11], on a single FPGA server.

REFERENCES

[1] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu, "Hyperscan: A fast multi-pattern regex matcher for modern cpus," 2019, p. 631–648. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/wang-xiang
[2] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100gbps intrusion prevention on a single server," 2020, p. 1083–1100. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng
[3] "Sigma - generic signature format for siem systems," Apr. 2026. [Online]. Available: https://github.com/SigmaHQ/sigma
[4] Z. Zhao, J. Melber, S. Sahay, S. Obla, E. Nurvitadhi, and J. C. Hoe, "Exploiting the common case when accelerating input-dependent stream processing by fpga," *IEEE Transactions on Computers*, vol. 72, no. 05, p. 1343–1355, May 2023.
[5] "AMD Vitis® HLS." [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html
[6] "High Level Synthesis Compiler — Quartus® Prime Design Software." [Online]. Available: https://www.altera.com/products/development-tools/quartus-prime/hls-compiler
[7] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "Fpga hls today: Successes, challenges, and opportunities," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 4, pp. 51:1–51:42, Aug. 2022.
[8] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong, "Autodse: Enabling software programmers to design efficient fpga accelerators," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 27, no. 4, pp. 32:1–32:27, Feb. 2022.
[9] "Transparent-computing/readme.md at master · darpa-i2o/transparent-computing." [Online]. Available: https://github.com/darpa-i2o/Transparent-Computing/blob/master/README.md
[10] [Online]. Available: https://www.elastic.co/elasticsearch
[11] M. Ur Rehman, H. Ahmadi, and W. Ul Hassan, "Flash: A comprehensive approach to intrusion detection via provenance graph representation learning," in *2024 IEEE Symposium on Security and Privacy (SP)*, May 2024, p. 3552–3570, iSSN: 2375-1207. [Online]. Available: https://ieeexplore.ieee.org/document/10646725