Optimizing Network Resilience Using Domain-Specific Hardware Accelerator for Dynamic Programming

Ali Mazloum*
amazloum@email.sc.edu
University of South Carolina
Columbia, South Carolina, USA

Sergio Elizalde*
elizalds@email.sc.edu
University of South Carolina
Columbia, South Carolina, USA

Samia Choueiri*
choueiri@email.sc.edu
University of South Carolina
Columbia, South Carolina, USA

Elie Kfoury*
ekfoury@email.sc.edu
University of South Carolina
Columbia, South Carolina, USA

Jose Gomez†
jagomez@fortlewis.edu
Katz School of Business
Durango, Colorado, USA

Ali AlSabeh‡
ali.alsabeh@usca.edu
University of South Carolina Aiken
Aiken, South Carolina, USA

Jorge Crichigno*
jcrichigno@cec.sc.edu
University of South Carolina
Columbia, South Carolina, USA

Abstract

Dynamic programming accelerators (DPAs) are devices designed with an instruction set optimized for dynamic programming (DP) operations. DP is fundamental to solving complex networking problems, particularly those involving fault tolerance and routing under dynamic conditions. This paper explores the use of DPA to accelerate network resilience by implementing an optimal routing algorithm that efficiently identifies alternative paths in response to link failures. The system's performance is evaluated by comparing DPA implementation against conventional GPU-based and CPU-based solutions. Results show that DPA provides significant performance improvements, enabling faster recovery and improved robustness in dynamic network environments.

CCS Concepts

• **Networks** → *Programmable networks*; *In-network processing*; *Network monitoring*; • **Hardware** → *Emerging tools and methodologies.*

Keywords

DSA for Dynamic Programming, Parallel Programming, CUDA, DPX, SDN.

ACM Reference Format:
Ali Mazloum, Sergio Elizalde, Samia Choueiri, Elie Kfoury, Jose Gomez, Ali AlSabeh, and Jorge Crichigno. 2025. Optimizing Network Resilience Using Domain-Specific Hardware Accelerator for Dynamic Programming. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3731599.3767449

1 Introduction

Dynamic programming (DP) is a well-established optimization technique used to solve complex problems by decomposing them into smaller, overlapping subproblems. By storing and reusing the solutions to these subproblems, DP avoids redundant computations, significantly improving efficiency. It has been widely applied across domains such as sequence alignment in bioinformatics [19] and route optimization in computer networking [22]. However, as data volumes continue to grow and the demand for real-time responsiveness intensifies, even highly optimized DP implementations often become performance bottlenecks when executed on general-purpose central processing units (CPUs), which offer limited parallelism and are not well-suited for compute-intensive tasks with fine-grained dependencies [20].

To address these challenges, general-purpose graphics processing units (GPGPUs) have been leveraged to accelerate DP workloads [1, 10, 23]. GPUs provide massive parallelism and high memory bandwidth, making them well-suited for data- and compute-intensive applications. Unlike CPUs, which are optimized for low-latency, sequential tasks, GPUs achieve high throughput by distributing work across thousands of lightweight cores. This parallel architecture allows many DP subproblems to be solved concurrently, significantly reducing execution time. Building on this capability, NVIDIA introduced Dynamic Programming eXecution (DPX) in its Hopper architecture. Those dynamic programming accelerators (DPAs) utilize a set of hardware-accelerated instructions designed specifically to accelerate DP workloads by offloading key operations to the GPU with reduced overhead and improved efficiency [11].

This paper focuses on accelerating network resilience using DPAs. As network size and complexity grow, the frequency of link and node failures increases [14, 15]. When such failures occur, the network enters a reconvergence phase to re-establish affected paths. During this period, some traffic may lack valid forwarding information, resulting in temporary loss of end-to-end connectivity and degraded performance, particularly for latency-sensitive, real-time applications [13]. Traditional control planes rely on routing protocols to recompute paths, but their convergence time, especially for remote failures, often exceeds 30 seconds [6]. Integrating DPAs into the control plane enables the rapid computation of alternative paths using parallel DP algorithms, offering a significant reduction in recovery time compared to GPGPU and CPU implementations.

This work investigates the use of DPAs to accelerate network failure recovery. The application is formulated as an all-pairs shortest path problem and implemented using the Floyd-Warshall algorithm. The implementation demonstrates how DPAs can be exploited to achieve substantial performance gains in latency-sensitive, compute-intensive networking functions.

The rest of the paper is organized as follows: Section 2 presents background information on the parallel programming, CUDA, and DPAs; Section 3 surveys related work; Section 4 describes the Floyd-Warshall algorithm and discusses the problem formulation and implementation of the network resilience application; Section 5 evaluates the proposed integration; and Section 6 concludes the paper.

2 Background on Parallel Programming, CUDA, and DPAs

The Single Instruction Multiple Threads (SIMT) paradigm extends the Single Instruction Multiple Data (SIMD) execution paradigm to multithreading, allowing multiple threads to execute the same instruction simultaneously. NVIDIA GPUs comprise multiple streaming multiprocessors (SMs), each containing SIMD cores. Each SM schedules and executes "warps," which are groups of 32 threads executing the same instruction and serve as the minimum scheduling unit [7]. The GPU memory hierarchy consists of several levels, each progressively smaller and faster. These levels include main memory, L2 cache, and L1/shared cache. Main memory and L2 cache are shared across all SMs, while each SM possesses its own on-chip L1/shared cache. The shared cache occupies the same physical space as the L1 cache but is software-managed by the programmer. Additionally, each thread has its own registers [17].

CUDA is an environment provided by NVIDIA that enables programmers to write GPU code using extensions of C/C++ or other languages [16]. It offers abstractions for thread group hierarchies, shared memories, and barrier synchronization. Programmers define "kernels" in CUDA, which are functions that execute in parallel on the GPU with a specified number of threads. CUDA provides the libraries to utilize DPX. DPX is an extension of the instruction set architecture (ISA) that enables DPAs. DPAs are programmed using low-level intrinsic functions for version 12 of the CUDA compiling toolset. They perform 2-operand and 3-operand maximum and minimum operations, as well as addition and maximum fused into a single instruction, with optional clamping to zero (relu variant). Operands can be either signed or unsigned integers. Specific hardware support for these functions is available only for GPU architectures with a compute capability equal to or greater than 9.0.

3 Related Work

GPGPUs have been explored as accelerators for DP workloads. One significant contribution in this area is the work of O'Connell [18], who proposed a generic GPU-based parallel model for dynamic programming. The model utilized an anti-diagonal (wavefront) traversal strategy to preserve data dependencies while enabling concurrent computation. It incorporated memory and thread optimizations to efficiently solve problems such as the knapsack and traveling salesman problems. While the approach achieved notable speedups over CPU implementations, it required manual tuning and emulated DP operations through standard GPU instructions without hardware-level acceleration.

While general models provide broad applicability, many researchers have focused on highly optimizing GPU implementations for specific, computationally intensive DP algorithms. For example, parallel implementations of the Floyd–Warshall algorithm have been proposed using block-wise decomposition and tiling strategies to optimize memory access patterns [5, 8, 21]. Similarly, optimized GPU-based versions of the Smith–Waterman algorithm have exploited the inherent parallelism of the scoring matrix to accelerate sequence alignment tasks [3]. These approaches offer high performance for their respective problems but are often limited in generalizability and lack support for low-level DP primitives.

Beyond GPGPUs, the pursuit of even higher performance and energy efficiency for DP kernels has led to the development of custom hardware accelerators, including Application-Specific Integrated Circuits (ASICs) [9, 12] and Field-Programmable Gate Arrays (FPGAs) [2]. These domain-specific accelerators are tailored to the unique computational patterns of particular DP problems, especially prevalent in genomics. For instance, frameworks like GenDP [4] offer specialized DP accelerators (DPAx) and sophisticated mapping algorithms (DPMap) that can adapt to various dependency patterns, objective functions, and precision requirements encountered in next-generation DNA sequencing pipelines. Luo et al. [11] conducted one of the first in-depth evaluations of the Hopper architecture, benchmarking DPAs alongside tensor cores and shared memory features. Their results demonstrated that DPAs substantially improve throughput and reduce latency for DP-heavy kernels compared to conventional CUDA-based implementations.

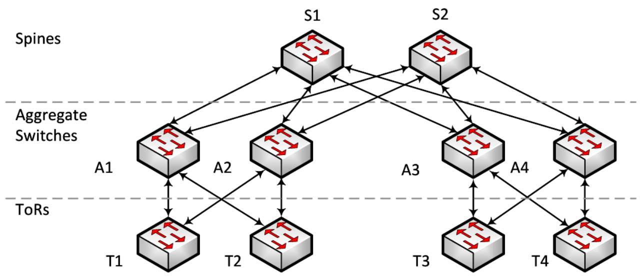
Despite these hardware advancements, the integration of DPAs into domain-specific applications, particularly in networking, has remained largely unexplored. Existing literature has not systematically applied DPAs to accelerate networking functions such as failure recovery. The present work addresses this limitation by demonstrating how DPAs can be used to accelerate critical networking workloads.

4 Network Resilience

This section introduces the Floyd-Warshall algorithm and formulates the network resilience as an all-pairs shortest path problem before describing the CUDA implementation.

4.1 Floyd-Warshall Algorithm

Floyd-Warshall is used to compute the shortest paths between all pairs of vertices in a weighted graph. It efficiently identifies the minimum cost to travel from any vertex i to any vertex j, even in the presence of negative edge weights (as long as there are no negative cycles). Let V be the set of vertices, with $|V| = n$, and let $w(i, j)$ be the weight of the edge from vertex i to vertex j, or ∞ if no such edge exists. Define a matrix D of size $n \times n$, where $D_{i,j}^{(k)}$



	T1	T2	T3	T4	A1	A2	A3	A4	S1	S2
T1	0	∞	∞	∞	2	3	∞	∞	∞	∞
T2	∞	0	∞	∞	3	2	∞	∞	∞	∞
T3	∞	∞	0	∞	∞	∞	2	3	∞	∞
T4	∞	∞	∞	0	∞	∞	3	2	∞	∞
A1	3	4	∞	∞	0	∞	∞	∞	3	4
A2	4	3	∞	∞	∞	0	∞	∞	4	3
A3	∞	∞	3	4	∞	∞	0	∞	3	4
A4	∞	∞	4	3	∞	∞	∞	0	4	3
S1	∞	∞	∞	∞	1	2	1	2	0	∞
S2	∞	∞	∞	∞	2	1	2	1	∞	0

$D^{(0)}$

Figure 1: Leaf-spine topology with the corresponding $D^{(0)}$ matrix.

represents the shortest distance from vertex i to vertex j using only the first k intermediate vertices (i.e., $\{1, 2, \ldots, k\}$).

The initial values of the matrix are defined as:

$$D_{i,j}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } i \neq j \end{cases}$$

Then, for each $k = 1$ to n, the matrix is updated according to the following recurrence:

$$D_{i,j}^{(k)} = \min\left(D_{i,j}^{(k-1)},\ D_{i,k}^{(k-1)} + D_{k,j}^{(k-1)}\right) \quad (1)$$

This recurrence expresses that the shortest path from i to j using vertices $\{1, \ldots, k\}$ is either the previously known shortest path (without using vertex k), or a new path that goes through k.

After completing all iterations, $D_{i,j}^{(n)}$ contains the length of the shortest path from i to j. If $D_{i,i} < 0$ for any i, the graph contains a negative cycle. To reconstruct the shortest paths, a predecessor matrix can be maintained during the computation and used to perform a traceback through the solution space.

4.2 Problem Formulation

Consider a network topology represented as a graph, where $V = \{v_0, v_1, \ldots, v_n\}$ denotes the set of networking devices (e.g., routers and switches), and $E = \{e_{i,j} \mid v_i, v_j \in V\}$ is the set of weighted edges, where each $e_{i,j}$ denotes the cost (e.g., delay or distance) of the link between nodes v_i and v_j.

Let $D^{(n)}$ be the distance matrix of size $n \times n$, where each entry $D_{i,j}^{(n)}$ represents the shortest path cost from node v_i to node v_j. Additionally, define a predecessor matrix $P^{(n)}$, also of size $n \times n$,

Algorithm 1: Serialized Floyd-Warshall Algorithm

```
Require: Vertex set $V = \{v_0, v_1, \ldots, v_n\}$, edge set $E = \{e_{i,j}\}$
1: for each intermediate vertex $v_k \in V$ do
2:     for each source vertex $v_i \in V$ do
3:         for each destination vertex $v_j \in V$ do
4:             $D_{i,j} \leftarrow \min(D_{i,j},\ D_{i,k} + D_{k,j})$
5:         end for
6:     end for
7: end for
8: return $D^{(n)}, P^{(n)}$
```

where each entry $P_{i,j}^{(n)}$ stores the immediate predecessor of node v_j on the shortest path from v_i to v_j.

Upon a node failure of $v_f \in V$, the vertex set is updated as $V \leftarrow V \setminus \{v_f\}$, and the edge set is updated by removing all edges connected to the failed node: $E \leftarrow E \setminus \{e_{i,f}, e_{f,i} \mid v_i \in V\}$. Similarly, in the case of a link failure between nodes v_i and v_j, the edge set is updated as $E \leftarrow E \setminus \{e_{i,j}, e_{j,i}\}$.

In both failure scenarios, the updated topology requires recomputation of the shortest path cost matrix $D^{(n)}$ and the predecessor matrix $P^{(n)}$, in order to reflect the new optimal paths among all remaining nodes in the network.

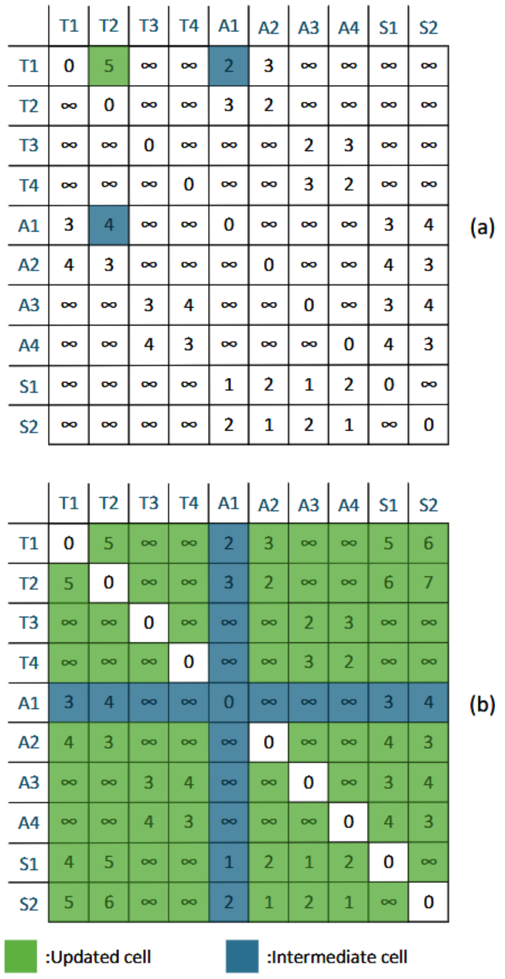
4.3 CUDA Implementation

Parallelized workload: As described in Algorithm 1, the serialized Floyd-Warshall algorithm consists of three nested loops, each iterating over the vertex set V. In each iteration of the outer loop, a single vertex $v_k \in V$ is selected as the intermediate node. The inner two loops traverse all source-destination pairs $(v_i, v_j) \in V \times V$, and update the entry $D_{i,j}$ by evaluating whether the path $v_i \longrightarrow v_k \longrightarrow v_j$ yields a shorter distance than the current value of $D_{i,j}$, in accordance with Equation 1. Each update relies on the values of $D_{i,k}$, $D_{k,j}$, and the current $D_{i,j}$.

Because all updates to $D_{i,j}$ for a fixed intermediate vertex v_k are independent, the computations for each cell in the distance matrix can be performed in parallel. In the CUDA implementation, the inner two loops are replaced by a GPU kernel that evaluates the entire $D^{(k)}$ matrix in a single launch. If the total number of available GPU threads exceeds n^2, where $n = |V|$, a one-to-one mapping is used where each thread computes a single cell $D_{i,j}$. Otherwise, each thread is assigned to compute approximately $n^2/\texttt{thread\_count}$ cells.

This is illustrated in Figs. 1 and 2. Figure 1 depicts a leaf-spine topology with four top-of-rack (ToR) switches, four aggregate switches, and two spine switches. The corresponding weighted adjacency matrix $D^{(0)}$ is shown below the topology. The weight between nodes with no direct connection is set to infinity. Figure 2 demonstrates the difference between the serialized and parallel approaches for a single iteration using node A1 as the intermediate vertex. In the serialized case, only one cell ($D_{\mathrm{T1,T2}}$) is updated by checking whether A1 provides a shorter path. In contrast, the parallel version updates all cells concurrently, evaluating A1 as an intermediate vertex for every node pair.

Coalesced memory access: Parallelizing workloads across available GPU threads is essential, but not sufficient, to fully exploit


(a)
(b)
:Updated cell
:Intermediate cell

(a)

	T1	T2	T3	T4	A1	A2	A3	A4	S1	S2
T1	0	5	∞	∞	2	3	∞	∞	∞	∞
T2	∞	0	∞	∞	3	2	∞	∞	∞	∞
T3	∞	∞	0	∞	∞	∞	2	3	∞	∞
T4	∞	∞	∞	0	∞	∞	3	2	∞	∞
A1	3	4	∞	∞	0	∞	∞	∞	3	4
A2	4	3	∞	∞	∞	0	∞	∞	4	3
A3	∞	∞	3	4	∞	∞	0	∞	3	4
A4	∞	∞	4	3	∞	∞	∞	0	4	3
S1	∞	∞	∞	∞	1	2	1	2	0	∞
S2	∞	∞	∞	∞	2	1	2	1	∞	0

(b)

	T1	T2	T3	T4	A1	A2	A3	A4	S1	S2
T1	0	5	∞	∞	2	3	∞	∞	5	6
T2	5	0	∞	∞	3	2	∞	∞	6	7
T3	∞	∞	0	∞	∞	∞	2	3	∞	∞
T4	∞	∞	∞	0	∞	∞	3	2	∞	∞
A1	3	4	∞	∞	0	∞	∞	∞	3	4
A2	4	3	∞	∞	∞	0	∞	∞	4	3
A3	∞	∞	3	4	∞	∞	0	∞	3	4
A4	∞	∞	4	3	∞	∞	∞	0	4	3
S1	4	5	∞	∞	1	2	1	2	0	∞
S2	5	6	∞	∞	2	1	2	1	∞	0

Figure 2: Shortest distance matrix after performing a single iteration using node A1 as the intermediate vertex. (a) represents the serialized approach. (b) represents the parallel approach.

the performance potential of GPUs. The pattern by which threads access global memory plays a crucial role in determining overall efficiency. Maximum performance is achieved when threads follow a coalesced memory access pattern, where all threads in a warp access contiguous memory locations. This enables the GPU to combine multiple memory accesses into a single transaction, significantly reducing memory latency and maximizing bandwidth utilization. As a result, coalesced access is essential for high-throughput, data-parallel applications.

In the context of the Floyd-Warshall algorithm, there are at least two approaches to ensure coalesced memory access. The first method employs a nested loop within the kernel: the outer loop iterates over the rows of the distance matrix D, while the inner loop iterates over its columns. The outer loop starts at the block ID and increments by the grid size, whereas the inner loop starts at the thread ID and increments by the block size. This layout ensures that threads within a warp access contiguous memory cells, achieving optimal coalescence.

The second method uses a single loop, where each thread iterates over the entire matrix using its global thread ID as the starting index and steps through the matrix by the total number of threads. This approach requires one modulus and one division operation to compute the row and column indices needed for evaluating Equation 1. While this method also achieves coalesced memory access, the arithmetic operations introduce slight computational overhead. In this work, the first method is adopted due to its simplicity and efficiency.

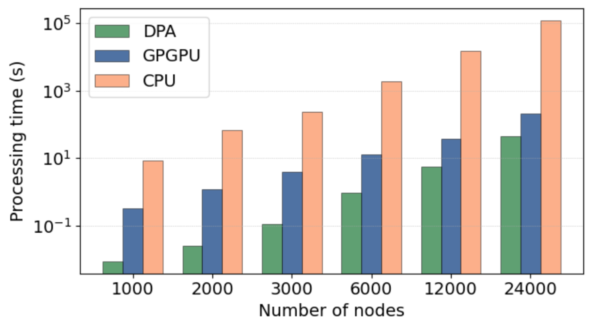


Figure 3: Processing time for different problem sizes on the DPA, the GPGPU, and CPU.

5 Experimental Results

This section presents the experimental evaluation of accelerating network resilience using DPAs. The performance is assessed across three dimensions: processing time, computational throughput (measured in billion cell updates per second, BCUPS), and energy consumption, over a range of problem sizes. Additionally, the impact of coalesced memory access on GPU performance is examined. The evaluation considers three platforms: the NVIDIA H100 (DPA), the NVIDIA A100 (GPGPU), and Intel Xeon Silver 4114 (CPU).

5.1 Processing Time

This experiment evaluates the processing time of the three platforms using input graphs with 1K, 2K, 3K, 6K, 12K, and 24K vertices. As shown in Figure 3, both GPUs consistently outperform the CPU across all problem sizes. The GPGPU achieves a speedup of 25× over the CPU with 1K nodes and up to 568× with 24K nodes. In contrast, the DPA shows a different trend. While it maintains a significantly faster performance than the CPU, with speedups ranging from 962× to 2642×, its performance advantage over the GPGPU decreases as the problem size increases. The speedup over the GPGPU declines from 45× at 2K nodes to 4× at 24K nodes.

For instance, with 1K nodes, the DPA computes all-pairs shortest paths in 8 milliseconds, which is 37× faster than the GPGPU (334 milliseconds) and 963× faster than the CPU (8.6 seconds). For mid-sized graphs ranging from 2K to 6K nodes, the DPA is the only platform achieving sub-second runtimes. For larger topologies, the CPU becomes impractical. With 24K nodes, the CPU requires 33

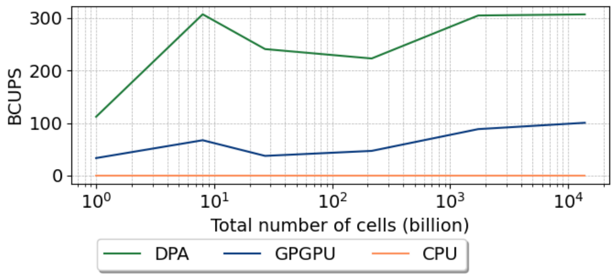


Figure 4: BCUPS achieved by the DPA, the GPGPU, and the CPU for different problem sizes.

hours, while the GPGPU and DPA complete the computation in 209 seconds and 45 seconds, respectively.

5.2 BCUPS

Figure 4 presents the achieved BCUPS, calculated as the total number of cell updates ($|V|^3$) divided by the processing time. The CPU maintains a constant rate of approximately 0.116 BCUPS across all problem sizes, showing no scalability or efficiency improvement with increased workload.

The the GPGPU exhibits a general upward trend in BCUPS as the problem size increases, although its performance varies. For consistency, the reported GPGPU values represent the average of five independent runs. The DPA consistently achieves higher throughput than both the GPGPU and CPU. For a graph with 1K nodes (10^9 cells), the DPA reaches 111.85 BCUPS, its lowest observed value, due to limited parallelism (only around 10^6 cells can be updated concurrently). At 8×10^9 cells, the DPA achieves 306.5 BCUPS, exceeding GPGPU by approximately 300 BCUPS. As the total number of cells increases to 13.824×10^{12}, the gap narrows to approximately 240 BCUPS.

5.3 Energy Consumption

This experiment measures the energy consumed by each platform using the NVIDIA NVML library for the GPUs and intel-rapl for

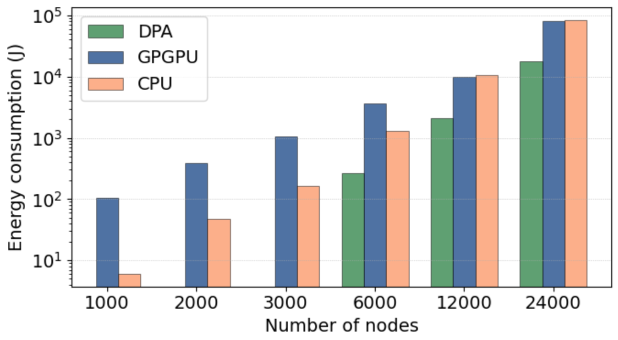


Figure 5: Energy consumption of the DPA, the GPGPU, and the CPU across different problem sizes.

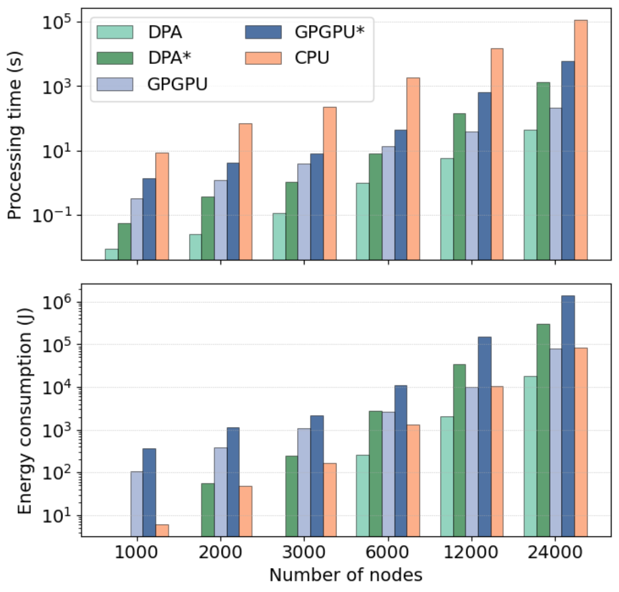


Figure 6: Impact of non-coalesced memory access on the processing time and energy consumption of GPU-based platforms. The * symbol indicates non-coalesced implementations.

the CPU. Energy consumption is calculated as the product of average power usage and execution time. Due to limited granularity in measurement, energy values for problem sizes with kernel execution times in the range of tens of milliseconds are unreliable. Consequently, values for 1K, 2K, and 3K nodes are omitted from Figure 5.

The DPA consistently demonstrates the highest energy efficiency. For example, with 6K nodes, it consumes 261.7 J, compared to 3584.1 J for the GPGPU and 1317.1 J for the CPU. This corresponds to a reduction of 13.7× and 5×, respectively. For larger problem sizes, the DPA maintains an energy advantage of approximately 4.6× to 4.8× over both the GPGPU and the CPU. For smaller sizes, the CPU outperforms the GPGPU in energy efficiency. At 1K nodes, the CPU consumes only 6 J, which is 17.5× less than the energy consumed by the GPGPU. As the problem size increases to 12K and 24K nodes, the CPU becomes less efficient than the GPGPU.

5.4 Impact of Coalesced Memory Access

This experiment assesses the effect of coalesced memory access on the processing time and energy consumption of the GPUs. Figure 6 and Table 1 show the performance degradation when memory access is not coalesced.

Both the DPA and the GPGPU experience significant increases in processing time without coalesced memory access. For the DPA, the processing time rises from (0.00894, 0.0261, 0.1123, 0.9706, 5.68, 45.13) seconds to (0.054, 0.379, 1.088, 7.975, 147.146, 1348.69) seconds across the six problem sizes. This change reflects a slowdown ranging from 6× to 29×. A similar pattern is observed on the GPGPU,

Table 1: Comparison of processing time and energy consumption for the DPA-enabled GPU and GPGPU with and without coalesced memory access. Processing time is calculated in seconds, and energy consumption is calculated in joules.

	Coalesced				Non-coalesced			
	DPA		GPGPU		DPA		GPGPU	
Number of nodes	Proc time	Energy consumption	Proc time	Energy consumption	Proc time	Energy consumption	Proc time	Energy consumption
1,000	0.00894	\-	0.334	105.8	0.054	\-	1.366	372.731
2,000	0.0261	\-	1.18	385.7	0.379	55.113	4.292	1138.727
3,000	0.1123	\-	3.849	1060.206	1.088	244.122	8.117	2126.465
6,000	0.9706	261.7	13.279	3584.13	7.975	2765.698	44.383	11230.983
12,000	5.68	2112.7	38.44	9970.143	147.146	34039.106	665.714	153713.777
24,000	45.13	17788.914	209.501	80202.223	1348.69	306594.25	6189	1407886

where the processing time increases from (0.334, 1.18, 3.849, 13.279, 38.44, 209.501) seconds to (1.366, 4.292, 8.117, 44.383, 665.714, 6189) seconds. The largest impact is recorded at 24K nodes, where the BCUPS for the DPA drops from 306.3 to 10.2, and for the GPGPU from 65.9 to 2.2.

Energy consumption increases similarly. In the non-coalesced case, the CPU becomes the most energy-efficient platform. For instance, with 6K nodes, the CPU consumes 1317.1 J, compared to approximately 2× and 8.5× more energy consumed by the DPA and the GPGPU, respectively. For 24K nodes, the CPU remains the most efficient, with 83428.299 J consumed, which is 3.6× and 16.8× less than the DPA and the GPGPU, respectively.

6 Conclusion

This paper evaluated the use of DPAs to accelerate dynamic programming algorithms for network resilience, using the Floyd–Warshall algorithm as a case study. Experimental results demonstrated that DPAs significantly outperformed both GPGPUs and CPUs in terms of processing time, throughput (BCUPS), and energy efficiency. The DPAs achieved sub-second runtimes for medium-sized networks and up to 2642× speedup over the CPU, while also consuming significantly less energy. The results also highlighted the critical role of coalesced memory access in achieving optimal GPU performance. These findings suggest that integrating DSAs into control planes can significantly enhance the responsiveness of failure recovery mechanisms in large-scale networks, providing a viable path toward more robust and efficient network infrastructures.

Acknowledgments

This work was supported by U.S. National Science Foundation award 2346726.

References

[1] Karl-Eduard Berger and Francois Galea. 2013. An efficient parallelization strategy for dynamic programming on GPU. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 1797–1806.

[2] Sergio Elizalde, Ali AlSabeh, Ali Mazloum, Samia Choueiri, Elie Kfoury, Jose Gomez, and Jorge Crichigno. 2025. A survey on security applications with smartnics: Taxonomy, implementations, challenges, and future trends. *Journal of Network and Computer Applications* (2025), 104257.

[3] Michael Farrar. 2006. Striped Smith–Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics* 23, 2 (11 2006), 156–161.

[4] Yufeng Gu, Arun Subramaniyan, Tim Dunn, Alireza Khadem, Kuan-Yu Chen, Somnath Paul, Md Vasimuddin, Sanchit Misra, David Blaauw, Satish Narayanasamy, et al. 2023. GenDP: A framework of dynamic programming acceleration for genome sequencing analysis. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[5] Pawan Harish and Petter J Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *International conference on high-performance computing*. Springer, 197–208.

[6] Thomas Holterbach, Stefano Vissicchio, Alberto Dainotti, and Laurent Vanbever. 2017. Swift: Predictive fast reroute. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 460–473.

[7] Minwook Je. 2024. [CUDA] Warps and SIMT. [Online]. Available: https://tinyurl.com/3jsx9aey.

[8] Gary J Katz and Joseph T Kider. 2008. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware (GH'08)*. 47–55.

[9] Elie F Kfoury, Samia Choueiri, Ali Mazloum, Ali AlSabeh, Jose Gomez, and Jorge Crichigno. 2024. A comprehensive survey on smartnics: Architectures, development models, applications, and research directions. *IEEE Access* (2024).

[10] Wouter Kool, Herke van Hoof, Joaquim Gromicho, and Max Welling. 2022. Deep policy dynamic programming for vehicle routing problems. In *International conference on integration of constraint programming, artificial intelligence, and operations research*. Springer, 190–213.

[11] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. 2024. Benchmarking and dissecting the NVIDIA Hopper GPU architecture. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE.

[12] Ali Mazloum, Ali AlSabeh, E Kfoury, and Jorge Crichigno. 2025. Domain Name Security Inspection at Line Rate: TLS SNI Extraction in the Data Plane Using P4 and DPDK. In *IEEE International Conference on Communications*, Vol. 8. 12.

[13] Ali Mazloum, Jose Gomez, Elie Kfoury, and Jorge Crichigno. 2023. Enhancing perfsonar measurement capabilities using p4 programmable data planes. In *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. 819–829.

[14] Ali Mazloum, Elie Kfoury, Ali AlSabeh, Jose Gomez, and Jorge Crichigno. 2025. Enhancing visibility on a science DMZ with P4-perfSONAR. *Journal of Network and Computer Applications* 242 (2025), 104263.

[15] Ali Mazloum, Elie Kfoury, Jose Gomez, and Jorge Crichigno. 2023. A survey on rerouting techniques with P4 programmable data plane switches. *Computer Networks* 230 (2023), 109795.

[16] NVIDIA. 2020. CUDA C++ programming guide. *NVIDIA, July* (2020).

[17] NVIDIA. 2025. Tuning CUDA Applications for Hopper GPU Architecture. [Online]. Available: https://tinyurl.com/msdfc2ps.

[18] Jonathan F O'Connell. 2017. *A dynamic programming model to solve optimisation problems using GPUs*. Ph. D. Dissertation. Cardiff University.

[19] Bharath Reddy and Richard Fields. 2021. Multiple sequence alignment algorithms in bioinformatics. In *Smart Trends in Computing and Communications: Proceedings of SmartCom 2021*. Springer, 89–98.

[20] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 1617–1632.

[21] Edvin Teskeredžić, Kenan Karahodžić, and Novica Nosović. 2020. Comparison of the Non-Blocked and Blocked Floyd-Warshall Algorithm with Regard to Speedup and Energy Saving on an Embedded GPU. In *2020 19th INFOTEH*. 1–5.

[22] Xiong Wang, Qian Zhang, Jing Ren, Shizhong Xu, Sheng Wang, and Shui Yu. 2018. Toward efficient parallel routing optimization for large-scale SDN networks using GPGPU. *Journal of Network and Computer Applications* 113 (2018), 1–13.

[23] Zhaoxuan Zhu, Shobhit Gupta, Nicola Pivaro, Shreshta Rajakumar Deshpande, and Marcello Canova. 2021. A GPU implementation of a look-ahead optimal controller for eco-driving based on dynamic programming. In *2021 (ECC)*. IEEE.