НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
сентябрь–октябрь 2025 Том 25 № 5 http://ntv.ifmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTICS
September–October 2025 Vol. 25 No 5 http://ntv.ifmo.ru/en/
ISSN 2226-1494 (print) ISSN 2500-0373 (online)

НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

doi: 10.17586/2226-1494-2025-25-5-866-875

Accelerating and analyzing performance of shortest path algorithms on GPU using CUDA platform: Bellman-Ford, Dijkstra, and Floyd-Warshall algorithms

Deep Bodra[1]✉, Sushil Khairnar[2]

[1] Harrisburg University of Science and Technology, Harrisburg, 17101, USA
[2] Virginia Tech, Virginia, 24061, USA

[1] Deepbodra97@gmail.com✉, https://orcid.org/0009-0009-4173-2447
[2] sushilk@vt.edu, https://orcid.org/0009-0006-5192-0175

Abstract
The computational demands of the shortest path algorithms on large-scale graphs with millions of vertices and edges pose significant challenges for serial implementations, often requiring hours of execution time even on powerful CPUs. This paper evaluates Graphic Processing Units implementations of three fundamental shortest path algorithms — Bellman-Ford, Dijkstra, and Floyd-Warshall using NVIDIA CUDA platform. We implemented and compared multiple variants of each algorithm, starting with basic parallel approaches and applying various optimization techniques, including grid-stride loops, shared memory utilization, memory coalescing, and algorithm-specific enhancements such as flag-based early termination for Bellman-Ford and tiled computation for Floyd-Warshall. Our study provides performance analysis comparing different optimization strategies and their effectiveness across various graph datasets.

Keywords
GPU computing, CUDA platform, shortest path algorithms, parallel algorithms, graph algorithms, Bellman-Ford, Dijkstra, Floyd-Warshall, performance optimization

For citation: Bodra D., Khairnar S. Accelerating and analyzing performance of shortest path algorithms on GPU using CUDA platform: Bellman-Ford, Dijkstra, and Floyd-Warshall algorithms. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2025, vol. 25, no. 5, pp. 866–875. doi: 10.17586/2226-1494-2025-25-5-866-875

УДК 004.424.4

Ускорение и анализ производительности алгоритмов поиска кратчайшего пути на GPU с использованием платформы CUDA: алгоритмы Беллмана–Форда, Дейкстры и Флойда–Уоршелла

Дип Бодра[1]✉, Сушил Хайрнар[2]

[1] Гаррисбергский университет науки и технологий, Гаррисберг, 17101, США
[2] Технологический институт Вирджинии, Вирджиния, 24061, США

[1] Deepbodra97@gmail.com✉, https://orcid.org/0009-0009-4173-2447
[2] sushilk@vt.edu, https://orcid.org/0009-0006-5192-0175

Аннотация
Вычислительные требования к алгоритмам поиска кратчайшего пути на больших графах с миллионами вершин и ребер представляют собой значительную проблему для последовательных реализаций, часто требуя многочасового времени выполнения даже с помощью мощных процессоров. В работе выполнена оценка реализации на графических процессорах трех фундаментальных алгоритмов поиска кратчайшего пути: Беллмана–Форда, Дейкстры и Флойда–Уоршелла с использованием платформы NVIDIA CUDA. Проведено сравнение нескольких вариантов каждого алгоритма, от базовых параллельных подходов до специфических алгоритмов улучшения. Исследованы базовые методы оптимизации, включая циклы с шагом сетки, использование общей памяти, объединение памяти. Также выполнен анализ алгоритмов улучшения, таких как раннее завершение на основе флагов для алгоритма Беллмана–Форда и тайловые вычисления для алгоритма

Флойда–Уоршелла. В исследовании представлен анализ производительности, выполнено сравнение различных стратегий оптимизации и их эффективности на различных наборах графовых данных.

Ключевые слова
вычисления на GPU, платформа CUDA, алгоритмы поиска кратчайшего пути, параллельные алгоритмы, алгоритмы графов, Беллман–Форд, Дейкстра, Флойд–Уоршелл, оптимизация производительности

Ссылка для цитирования: Бодра Д., Хайрнар С. Ускорение и анализ производительности алгоритмов поиска кратчайшего пути на GPU с использованием платформы CUDA: алгоритмы Беллмана–Форда, Дейкстры и Флойда–Уоршелла // Научно-технический вестник информационных технологий, механики и оптики. 2025. Т. 25, № 5. С. 866–875 (на англ. яз.). doi: 10.17586/2226-1494-2025-25-866-875

Introduction

Shortest path algorithms have applications across domains including transportation networks, communication systems, social network analysis, and Very Large-Scale Integration chip design [1]. These algorithms solve the problem of finding the minimum cost path between vertices in weighted graphs, which is required in real-world scenarios such as GPS navigation systems to find optimal routes, network protocols to determine efficient data transmission paths, and circuit designers to optimize signal routing. The computational complexity of the problem becomes challenging as graph sizes grow to millions of vertices and edges, commonly encountered in modern applications, making serial implementations impractical for time-sensitive applications [1, 2].

Modern Graphics Processing Units (GPUs) contain many cores capable of executing operations in parallel which make them suitable for algorithms that can exploit data parallelism. However, parallelizing shortest path algorithms on GPU presents unique challenges including irregular memory access patterns, varying computational loads across threads, and complex data dependencies that can limit parallel efficiency [1, 2]. The inherent nature of shortest path algorithms can affect their suitability for GPU implementation. Bellman-Ford can handle negative edge weights but requires multiple iterations, Dijkstra provides better performance for non-negative weights but has inherent sequential dependencies, and Floyd-Warshall computes all-pairs shortest paths with high computational complexity [3].

This paper presents a comprehensive study of GPU implementations for three fundamental shortest path algorithms using Compute Unified Device Architecture (CUDA). We implement and evaluate multiple optimization strategies for each algorithm, progressing from basic parallel approaches to sophisticated techniques including memory optimization, algorithmic enhancements, and architecture-specific optimizations. Our evaluation focuses on understanding the performance characteristics and trade-offs of different implementation approaches across various graph datasets.

Background of the problem

The Bellman-Ford algorithm finds the shortest paths from a source vertex to all other vertices in a weighted graph [2]. The key advantage of Bellman-Ford over Dijkstra algorithm is its ability to detect negative cycles and handle graphs containing negative cycles reachable from the source vertex [4]. The algorithm performs a series of relaxation and iteratively improves distance estimates until optimal paths are found. For a graph with $|V|$ vertices, the algorithm performs $|V| - 1$ iterations, where each of iteration relaxes all edges and updates distance estimates if a shorter path is discovered [2]. Since the order of edge relaxation within iteration does not affect correctness, parallelization can be achieved by allowing multiple edges to be processed simultaneously.

When applied to the all-pairs shortest path problem, Bellman-Ford must be executed $|V|$ times, once for each source vertex. For number of edges $|E|$, the sequential time complexity becomes $O(|V|^2 \times |E|)$, as each of the $|V|$ source vertices require $O(|V| \times |E|)$ time for single-source computation.

The space complexity for storing all-pairs distances is $O(|V|^2)$ for the distance matrix, plus $O(|V| + |E|)$ for the graph representation, yielding total space complexity $O(|V|^2 + |E|)$. The answer size is $O(|V|^2)$ for distance values only, or $O(|V|^3)$ if complete path information is stored for all vertex pairs [2].

Dijkstra algorithm solves the single-source shortest path problem by maintaining a priority queue of vertices ordered by their current shortest distance estimate and greedily selects the vertex with minimum distance for processing. The serial version uses a min-heap to extract the closest unvisited vertex, and then relaxes all outgoing edges from that vertex. This inherently sequential process of selecting the next minimum vertex poses significant challenges for parallelization [1, 3]. However, parallel versions can be implemented by running multiple instances simultaneously, computing shortest paths from different source vertices. For all-pairs shortest path computation, Dijkstra algorithm must be executed $|V|$ times, once from each source vertex.

The sequential time complexity becomes $O(|V| \times (|V| + |E|) \log_2|V|)$, representing $|V|$ executions of single-source Dijkstra with $O((|V| + |E|) \log_2|V|)$ complexity each. The space complexity requires $O(|V|^2)$ for storing the complete distance matrix, plus $O(|V|)$ for the priority queue and temporary arrays during each execution, yielding total space complexity $O(|V|^2)$. The answer size is $O(|V|^2)$ for distance information, or $O(|V|^3)$ when complete shortest path trees are maintained for all source vertices [2].

The Floyd-Warshall's algorithm computes shortest paths between all pairs of vertices in a weighted graph [2]. It can handle negative edge weights but not negative cycles. The algorithm employs dynamic programming, considering each vertex as an intermediate point in potential shortest paths. The algorithm executes $|V|$ iterations, where iteration k considers vertex k as an intermediate vertex for all vertex pairs (i, j). For each pair, it checks whether the path $i \to k \to j$ offers a shorter distance than the current best path

from i to j [3]. This structure makes Floyd-Warshall highly suitable for parallelization, as all pairwise distance updates (within each iteration) can be performed independently. Floyd-Warshall is inherently designed for the all-pairs shortest path problem. The sequential time complexity is $O(|V|^3)$, as it performs $|V|$ iterations, each examining all $|V|^2$ vertex pairs for potential improvement through the current intermediate vertex. The space complexity is $O(|V|^2)$ for storing the distance matrix which directly represents the shortest distances between all vertex pairs. The answer size is exactly $O(|V|^2)$ for distance information, or $O(|V|^3)$ when complete path reconstruction information is maintained. Unlike Bellman-Ford and Dijkstra, Floyd-Warshall complexity does not scale with the number of edges $|E|$, making it particularly efficient for dense graphs where $|E|$ approaches $|V|^2$ [5, 6].

CUDA Computing

GPUs feature a massively parallel architecture designed for high-throughput computation[1]. Unlike CPUs with few powerful cores optimized for sequential processing, modern GPUs contain thousands of smaller cores that excel at executing the same operation across multiple data elements simultaneously. This Single Instruction, Multiple Data architecture makes GPUs particularly effective for data-parallel algorithms[1].

CUDA provides a programming framework for general-purpose GPU computing developed by NVIDIA. In CUDA, parallel work is organized into kernels — functions executed simultaneously by many threads. Threads are grouped into blocks, and blocks are organized into a grid structure. This hierarchical organization enables efficient resource management and communication patterns[1].

The CUDA programming model encompasses several key concepts relevant to this work. The thread hierarchy organizes individual threads that execute kernel code into blocks that can share memory and synchronize operations. The memory hierarchy provides different levels of storage including global memory with large capacity but high latency, shared memory that offers fast access but limited capacity per block, and registers that provide the fastest access but are limited per thread[1] Grid-stride loops represent a programming pattern that allows kernels to process datasets larger than the number of available threads. Atomic operations provide hardware-supported mechanisms for thread-safe memory updates, which are crucial for avoiding race conditions in parallel algorithms [1].

Challenges in GPU Graph Algorithm Implementation

Implementing graph algorithms on GPUs presents several significant challenges [3] that have been extensively documented in the literature. Graph traversal often results in irregular memory access patterns that can substantially reduce GPU efficiency, as the hardware is optimized for coalesced memory accesses [6, 7]. Load balancing presents another critical challenge, as vertices may have vastly different degrees, leading to uneven work distribution across threads and resulting in some threads completing their work much earlier than others [6, 7]. Synchronization requirements in many graph algorithms necessitate coordination between threads, which can potentially limit the achievable parallelism and introduce performance bottlenecks [1, 3]. Memory bandwidth limitations arise when large graphs exceed GPU memory capacity or create memory bandwidth bottlenecks that constrain overall performance [3]. Understanding these fundamental challenges is essential for developing effective GPU implementations of shortest path algorithms and forms the foundation for the optimization strategies explored in this work.

Related Works

Yang et al. [8] proposed a Fast APSP algorithm that combines Floyd-Warshall with Dijkstra algorithms for large sparse graphs, achieving an average speedup of 16.97 times compared to CPU Dijkstra and 7.09 times compared to GPU Dijkstra implementations. Their work addresses graphs with over 11 million vertices using 2048 GPUs, demonstrating scalability beyond single-GPU implementations. Prihozhy and Karasik[2] conducted comprehensive comparisons of competing all-pairs shortest path algorithms for both sparse and dense graphs, providing valuable insights into algorithm selection criteria. However, their approach requires distributed computing clusters, whereas our work focuses on optimizing single-GPU performance through tiling and shared memory techniques.

Tang et al. [9] developed GPU-accelerated all-pairs shortest path algorithms specifically for stochastic road networks, reporting “thousands of times improvement” in acceleration for real-world navigation applications. While their work targets similar applications to ours, their focus on stochastic networks with uncertainty handling differs from our deterministic graph optimization approach. Recent research has explored innovative GPU-based methods for related graph problems. Spridon et al. [10] introduced novel GPU-based approaches for the generalized maximum flow problem, demonstrating the continued evolution of GPU graph algorithm development. These advances in related graph problems inform optimization strategies applicable to shortest path algorithms.

Traditional GPU implementations of Bellman-Ford have focused on basic parallelization strategies. Agarwal and Dutta [11] introduced flag-based optimization for GPU Bellman-Ford, which serves as a foundation for our strided with flag implementation. However, recent literature shows limited advancement in Bellman-Ford GPU optimization techniques beyond basic parallelization patterns, indicating a gap that our comprehensive optimization analysis addresses.

Early GPU implementations by Harish and Narayanan [1] demonstrated the potential for GPU acceleration of graph algorithms, achieving significant speedups for

[1] CUDA C++ Programming Guide. Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/, free. English lang. (accessed: 02.06.2025).

[2] Prihozhy A.A., & Karasik O.N. (2024). Competing all-pairs shortest paths algorithms for sparse/dense graphs: implementation and comparison. Available at: http://dx.doi.org/10.21122/2309-4923-2024-4-4-12, free. English lang. (accessed: 06.06.2025).

single-source shortest path computation on graphs with millions of vertices. Recent work by Song [12] explored high-performance parallelization of Dijkstra algorithm using hybrid Message Passing Interface and CUDA approaches, demonstrating the continued relevance of GPU acceleration for shortest path problems. However, these implementations focused on single-source shortest paths, while our parallel all-pairs approach addresses the inherent limitations of parallelizing Dijkstra sequential vertex selection process.

Contemporary research has expanded GPU shortest path applications beyond traditional graph problems. Bengtsson et al. [13] applied GPU-accelerated routing to warehouse optimization problems, combining clustering and dynamic systems modeling with GPU-based shortest path computation. Kumar et al. [14] investigated Artificial Intelligent based navigation in quasi-structured environments, highlighting the growing importance of GPU-accelerated path planning in robotics and autonomous systems. These applications demonstrate the practical relevance of efficient GPU shortest path implementations across diverse domains.

The effectiveness of shared memory utilization in GPU graph algorithms has been demonstrated across multiple studies [6]. However, comprehensive analysis of the trade-offs between different memory optimization techniques for shortest path algorithms remains limited in recent literature. Our work contributes detailed performance analysis of shared memory vs. global memory approaches specifically for Floyd-Warshall tiled implementations.

Recent research has emphasized the importance of achieving performance portability across different computing architectures. Morgan et al. [15] investigated simplified approaches to achieve parallel performance and portability across CPU and GPU architectures, highlighting the challenges of maintaining efficiency across diverse hardware platforms. This work underscores the importance of architecture-specific optimizations like those explored in our study.

Harris [7] introduced grid-stride loops as a fundamental CUDA optimization pattern. While this technique has been applied to various GPU algorithms, systematic evaluation of its effectiveness for different shortest path algorithm variants has not been thoroughly investigated in recent literature.

Our Contribution

Our work advances the current state of GPU shortest path algorithm research through several distinct contributions that address gaps in the existing literature. Unlike recent studies that focus on individual algorithms or specific application domains, we provide a comprehensive systematic comparison across three fundamental shortest path algorithms using consistent experimental methodology and hardware platforms. This multi-algorithm approach enables direct performance comparisons and reveals algorithm-specific optimization opportunities that have not been thoroughly explored in recent literature.

We implement and evaluate multiple optimization strategies for each algorithm, progressing from basic parallel implementations to sophisticated techniques that combine algorithmic and architectural optimizations. This progressive optimization evaluation methodology provides detailed insights into the incremental effects of different optimization techniques which has been limited in recent publications that typically focus on single optimization approaches.

Our study contributes quantitative analysis of the effectiveness of specific optimization techniques, including flag-based early termination for Bellman-Ford, tiled computation for Floyd-Warshall, and shared memory utilization across algorithms. The systematic evaluation of these techniques provides performance trade-off analysis that has been absent from recent literature.

We demonstrate that optimal GPU implementation strategies vary significantly across different shortest path algorithms, providing practical guidance for algorithm selection and optimization in real-world applications. This algorithm-specific optimization insight fills a gap in current literature where optimization strategies are often presented as universally applicable without considering algorithm-specific characteristics.

While prior work has established the foundation for GPU shortest path algorithms, our comprehensive analysis of optimization strategies and their trade-offs provides novel insights that advance the current state of knowledge in this domain.

Experimental Setup

All experiments were conducted on an NVIDIA GeForce RTX 2080 Ti with CUDA compute capability 7.5, CUDA driver version 10.1, 4352 CUDA cores, and 11 GB GDDR6 memory. This section describes the algorithms and their variants that were used for benchmarking. For each algorithm, we implemented multiple variants with different optimization approaches, ranging from basic parallel implementations to more sophisticated techniques with memory optimizations and algorithmic enhancements. All implementation code is available at platform GitHub[1].

Experimental Setup: Bellman-Ford algorithm

For the Bellman-Ford algorithm evaluation, we used the DIMACS Road Networks Dataset[2], which provides real-world road network graphs suitable for single-source shortest path analysis. The dataset contains various road networks with different scales, allowing us to evaluate performance across graphs of varying sizes and densities. All Bellman-Ford variants use Compressed Sparse Row (CSR) format to efficiently handle large graphs in GPU global memory [2].

One thread per vertex: This implementation assigns one thread to each vertex, where each thread is responsible for relaxing all outgoing edges from its assigned vertex [1]. It uses two distance arrays: *previousDistance*

[1] CUDA Parallel Shortest Path: Implementation. Available at: https://github.com/deepbodra97/cuda-parallel-shortest-path, free. English lang. (accessed: 05.06.2025).

[2] 9th DIMACS Implementation Challenge: Shortest Paths. Available at: http://www.diag.uniroma1.it/~challenge9/download.shtml, free. English lang. (accessed: 03.06.2025).

stores costs from the previous iteration, while distance accumulates updated costs in the current iteration. This dual-array approach prevents threads from reading updated values within the same iteration to maintain algorithmic correctness. After each iteration, values are copied from distance to *previousDistance*. A challenge arises when multiple threads attempt to update the same destination vertex simultaneously, creating race conditions. We address this using CUDA *atomicMin* operation, which ensures thread-safe updates by atomically selecting the minimum value among competing writes[1]. The kernel is launched $|V| - 1$ times, with each launch handling one complete iteration of edge relaxations. The time complexity is $O((|V| - 1) \times (|E|/P + sync\_cost))$, where $P = \min(|V|, GPU\ cores)$, as each thread processes edges in parallel but synchronization is required between iterations. Space complexity remains $O(|V|^2 + |E|)$ for the all-pairs distance matrix and CSR graph representation.

Strided: The one thread per vertex implementation scalability is limited by the maximum number of threads a GPU device supports. The strided version overcomes this limitation using a grid-stride loop pattern [7], where each thread processes multiple vertices depending on the grid size and total vertex count. In this approach, a thread with ID *tid* processes vertices at positions *tid*, *tid* + *stride*, *tid* + + 2 × *stride*, and so forth, where stride equals *blockDimension* × *gridDimension*. This pattern allows the same kernel to handle graphs of arbitrary size by using fewer threads than vertices [1]. The optimal number of threads for a given graph can be determined experimentally, balancing resource utilization with memory bandwidth. Time complexity becomes $O((|V| - 1) \times (\lceil |V|/P \rceil \times \times avg\_degree + sync\_cost))$ where $avg\_degree = |E|/|V|$, as each thread now processes multiple vertices sequentially within each iteration. The load balancing factor ranges from $O(1)$ for uniform degree distribution to $O(max\_degreee/avg\_degree)$ for skewed distributions, maintaining the same $O(|V|^2 + |E|)$ space complexity.

Strided with flag [11]: Both previous implementations suffer from inefficient work distribution, as threads process vertices whose distances changed since the previous iteration. The flag-based optimization addresses this by tracking which vertices had distance updates in the previous iteration. We maintain a Boolean flag array of size $|V|$, where $flag[i]$ indicates whether vertex i--th distance changed in the previous iteration. During the distance update phase, if $previousDistance[i] > distance[i]$, we set $flag[i] = true$. In the subsequent iteration, threads only process outgoing edges from vertices with $flag[i] = true$, significantly reducing unnecessary computations [11]. The flag is reset to false when a vertex edges are processed, preparing for the next iteration. The time complexity per iteration i becomes $O(Active\_vertices(i) \times avg\_degree/P)$, where $Active\_vertices(i)$ represents vertices with updated distances. In the expected case, this yields $O(|E| \times H\_|V|/P)$ where $H\_|V|$ is the $|V|^{th}$ harmonic number. Space complexity remains $O(|V|^2 + |E|)$ plus $O(|V|)$ for the flag array.

[1] CUDA C++ Programming Guide. Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/, free. English lang. (accessed: 02.06.2025).

Experimental Setup: Dijkstra algorithm

For Dijkstra algorithm evaluation, we used Stanford's Peer-to-peer network dataset [16], which is well-suited for evaluating all-pairs shortest path computations in moderately-sized graphs. The graph is represented using an adjacency matrix format for efficient neighbor lookups during shortest path computation.

Dijkstra algorithm presents parallelization challenges due to its inherently sequential nature of selecting the minimum unvisited vertex [1, 3, 17]. Instead of parallelizing the core algorithm logic, we implement a parallel all-pairs approach where each thread computes shortest paths from a different source vertex. Each thread runs an instance of Dijkstra algorithm using a different source vertex. Since GPU threads cannot efficiently maintain individual priority queues due to memory constraints, each thread uses a simple array-based approach to find the next minimum unvisited vertex [3]. The kernel assigns one thread per source vertex, with thread *src* computing shortest paths from vertex *src* to all other vertices. Each thread maintains its own visited array and distance array within the global distance matrix. This approach achieves parallelism across different source vertices while preserving the sequential correctness of individual Dijkstra computations. Our approach executes $|V|$ independent Dijkstra instances in parallel, yielding time complexity $O((|V|^2 + |E|) \times |V|/P)$ where each thread performs $O(|V|^2 + |E|)$ work for its assigned source. Space complexity is $O(|V|^2 + |V| \times P)$ for the distance matrix and per-thread visited arrays, with optimal performance when $P = |V|$.

Experimental Setup: Floyd-Warshall algorithm

For the Floyd-Warshall algorithm evaluation, we used Stanford's Peer-to-peer network dataset [16] and the graph is represented using an adjacency matrix format.

One thread per edge: This implementation provides maximum parallelism by assigning one thread to each edge in the graph [9]. Using a 2D thread grid, thread (i, j) handles the edge from vertex i to vertex j. In each of the iterations $|V|$, every thread checks whether using the current intermediate vertex k provides a shorter path. This approach achieves excellent memory coalescing as threads in the same warp access consecutive memory locations in the distance matrix [10]. However, it requires launching $|V|^2$ threads, limiting scalability for very large graphs due to GPU resource constraints. Time complexity is $O(|V|^3/P)$, where $P = \min(|V|^2, GPU\ cores)$, as $|V|^2$ threads perform $O(1)$ work per iteration across $|V|$ iterations. This achieves optimal parallelization when sufficient GPU cores are available. Space complexity remains $O(|V|^2)$ for the distance matrix with optimal memory coalescing patterns

One thread per edge with shared memory: This variant optimizes the one thread per edge approach by reducing global memory accesses through shared memory [6, 18] utilization. In iteration k, threads in the same row all access $distance[i][k]$, creating an opportunity for shared memory optimization. We use 1D thread blocks where the first thread in each block loads $distance[i][k]$ into shared memory, making it available to all threads in the block through shared memory broadcast [6]. This reduces global memory bandwidth requirements, though synchronization overhead can limit performance gains. Time complexity

remains $O(|V|^3/P)$ but with reduced memory access latency due to shared memory utilization. Space complexity includes additional O (*shared_memory_per_block*) for cached data, though synchronization overhead may offset some performance benefits for smaller graphs.

One thread per vertex implementation: To improve scalability, the one thread per vertex implementation reduces thread count by assigning one thread per vertex rather than per edge. Each thread handles one row of the distance matrix, iterating through all potential destinations for its assigned source vertex. Thread i processes all edges from vertex i, checking each destination j to determine if routing through intermediate vertex k offers improvement. This approach requires fewer threads while maintaining reasonable parallelism, making it suitable for larger graphs. Time complexity becomes $O(|V|^3/P)$, where $P = \min(|V|,$ *GPU cores*), but each thread now performs $O(|V|^2)$ work across all iterations. This provides better scalability for large graphs by reducing thread count requirements while maintaining the same $O(|V|^2)$ space complexity for the distance matrix.

Tiled implementation: This implementation addresses memory bandwidth limitations by dividing the adjacency matrix into 2D tiles of size *TILE_DIMENSION* × *TILE_DIMENSION*. At each iteration, tiles are processed in three phases. In phase 1, the primary tile (diagonal tile containing intermediate vertex) is processed using a single thread block, where all paths within this tile are updated using vertices from the same tile as intermediates. In phase 2, tiles sharing the same row or column as the primary tile are processed, where these tiles use vertices from the primary tile as intermediates, requiring data from both the primary tile and the current tile. In phase 3, the remaining tiles are processed using precomputed results from primary row and column tiles. This phase achieves maximum parallelism as all remaining tiles can be processed independently [5, 19, 20]. This approach ensures that each global memory location is accessed exactly once per iteration, significantly improving memory efficiency compared to the basic implementations. The time complexity becomes $O(|V|^2 + |V| \times$ *TILE_DIMENSION*) through the three-phase approach: primary tile $O(|V|)$, border tiles $O(|V|)$, and interior tiles O (*TILE_DIMENSION*) with full parallelization [5]. Space complexity remains $O(|V|^2)$ with optimal tile size *TILE_DIMENSION* = $\sqrt{}$(*GPU cores*) minimizing total execution time.

Tiled with shared memory: This implementation enhances the basic tiled approach by utilizing shared memory [6] to cache frequently accessed data within each thread block. In phase 1, the primary tile data is loaded into shared memory once and reused for all computations within the tile, eliminating redundant global memory accesses. During phase 2, each thread block loads the relevant portion of the primary tile into shared memory alongside its tile data, enabling fast access to intermediate vertex information. In phase 3, thread blocks load data from their corresponding row and column tiles into shared memory, significantly reducing global memory bandwidth requirements as multiple threads access the same cached values. Time complexity improves to $O(|V|^2 + |V| \times$ $\times$ *TILE_DIMENSION*/*memory_speedup*), where *memory_speedup* ≈ 10–100 times from shared memory utilization [6]. Space complexity includes $O(|V|^2 + \min($*num_blocks* × *TILE_DIMENSION*^{2}, *total_shared_memory*$))$ for the distance matrix plus shared memory usage, representing the most optimized implementation combining algorithmic restructuring with memory hierarchy optimization.

Results and Analysis

The Bellman-Ford algorithm experiments were conducted on the DIMACS Road Networks dataset[1], with performance measured across different graph sizes. Fig. 1 shows the execution times for the three implementation variants across various datasets. The one thread per vertex implementation serves as the baseline, providing straightforward parallelization but facing scalability limitations for larger graphs.

The strided implementation shows mixed performance results: it runs slower than the baseline for small graphs due to the overhead of grid-stride loops [7], but demonstrates better scalability for larger graphs where the naive version becomes resource-constrained.

The strided with flag optimization delivers the most significant performance improvements, achieving approximately 2.8 times faster execution compared to the one thread per vertex implementation. This optimization proves highly effective because it eliminates unnecessary work by processing only vertices whose distances changed in the previous iteration [11]. For the largest dataset (Eastern USA with 3.6 million vertices and 8.8 million edges), the strided with flag variant completed in 409 s compared to 1137 s for the baseline implementation. The performance characteristics reveal that the flag optimization becomes increasingly beneficial as graph size grows, since larger graphs tend to have more vertices with unchanged distances in later iterations of the algorithm.

Dijkstra algorithm evaluation used Stanford's Peer-to-peer network dataset [16]. Table shows the performance comparison between CPU and GPU implementations. The parallel all-pairs GPU implementation achieved significant speedup over the serial CPU version, completing the p2p-Gnutella04 dataset with 11K vertices and 40K edges in approximately 120 s compared to 24 minutes for the CPU implementation. However, the inherently sequential nature of Dijkstra algorithm limits the parallelization benefits compared to the other algorithms studied [1, 3].

The GPU implementation performance is constrained by the need for each thread to maintain its distance and visited arrays, along with the sequential process of finding minimum unvisited vertices within each thread computation. Despite these limitations, the GPU version still provides meaningful acceleration for all-pairs shortest path computation.

The Floyd-Warshall algorithm experiments demonstrate the most performance improvements among the three algorithms studied. Fig. 2 illustrates the execution times across different implementation variants and dataset sizes.

[1] 9th DIMACS Implementation Challenge: Shortest Paths. Available at: http://www.diag.uniroma1.it/~challenge9/download.shtml, free. English lang. (accessed: 03.06.2025).

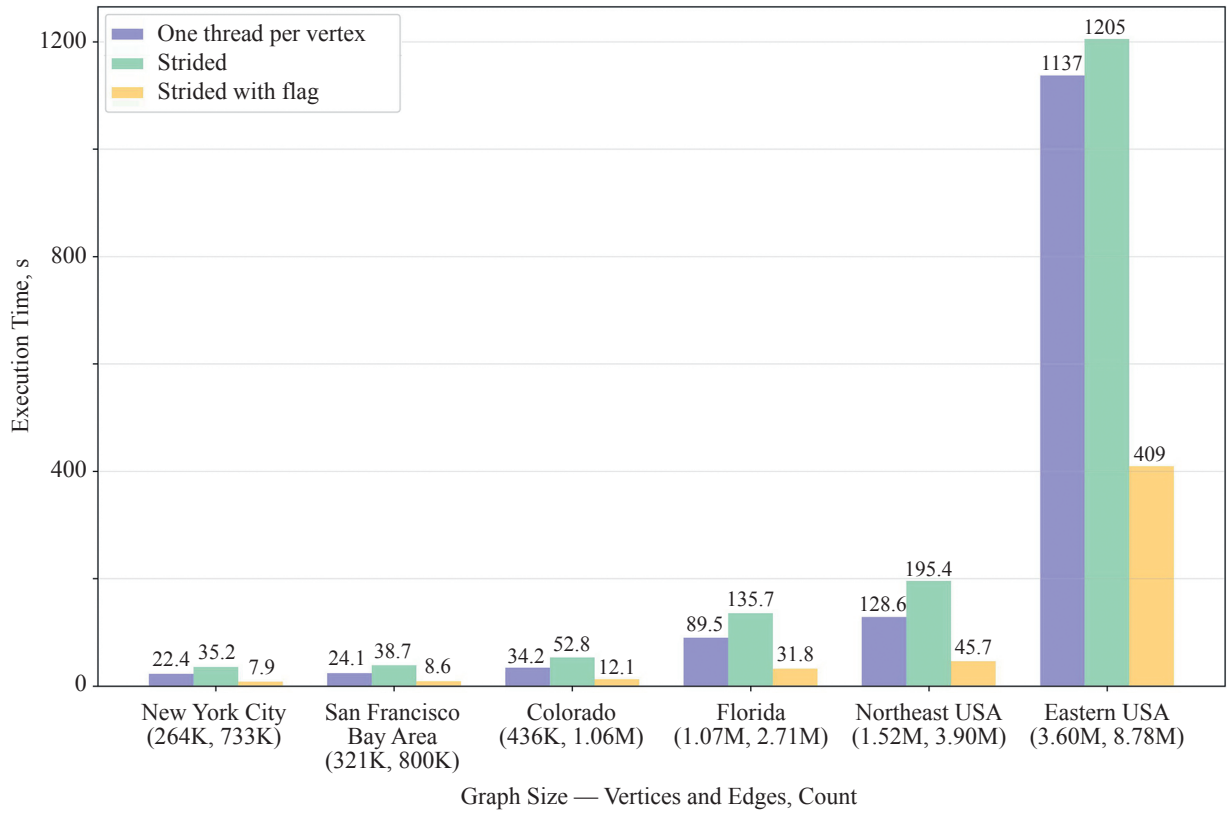

Fig. 1. Execution time comparison for Bellman-Ford variants on DIMACS Road Networks Dataset

Table. Execution time comparison for Dijkstra on Stanford's Peer-to-peer network dataset

Dataset	Number of Vertices	Number of Edges	CPU	GPU
p2p-Gnutella04	10,876	39,994	24 mins	119.941 s

The one thread per edge implementation achieves excellent performance for small to medium graphs due to maximum parallelization and good memory coalescing[1]. However, its scalability is limited by the requirement for $|V|^2$ threads. The variant of one thread per edge implementation with shared memory shows some improvement through reduced global memory accesses [6], though synchronization overhead can limit gains.

The one thread per vertex implementation provides better scalability by reducing thread count, making it suitable for larger graphs while maintaining reasonable performance. However, the tiled implementations show the most significant improvements [5].

The tiled implementation using global memory outperforms all basic variants, running approximately 4–6 times faster than the one thread per vertex version. The tiled implementation with shared memory delivers the best overall performance, achieving roughly 8 times speedup over the baseline and approximately 2 times improvement over the global memory tiled version [6].

For the p2p-Gnutella04 dataset, the tiled implementation with shared memory completed in 2.38 s compared to over 30 s for the one thread per vertex variant. This improvement demonstrates the effectiveness of combining algorithmic restructuring (tiling) with memory hierarchy optimization (shared memory) [5, 6].

When comparing across algorithms, Floyd-Warshall shows the greatest potential for GPU acceleration due to its inherently parallel structure and regular memory access patterns [17]. The algorithm $O(|V|^3)$ complexity makes GPU acceleration particularly valuable for reducing computation time.

Bellman-Ford demonstrates good parallelization potential, especially with the flag optimization that reduces unnecessary work [11]. The algorithm iterative nature and edge-based parallelism translate well to GPU architectures [1, 2].

Dijkstra algorithm shows the most limited parallelization benefits due to its inherently sequential vertex selection process [1, 3]. However, the all-pairs parallel approach still provides meaningful acceleration over serial CPU implementations.

The results highlight the importance of algorithm-specific optimizations: flag-based early termination for Bellman-Ford [11], tiled computation with shared memory for Floyd-Warshall [5, 6], and parallel source processing for Dijkstra. Memory access patterns and work distribution significantly impact GPU performance, with regular access

[1] CUDA C++ Programming Guide. Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/, free. English lang. (accessed: 02.06.2025).

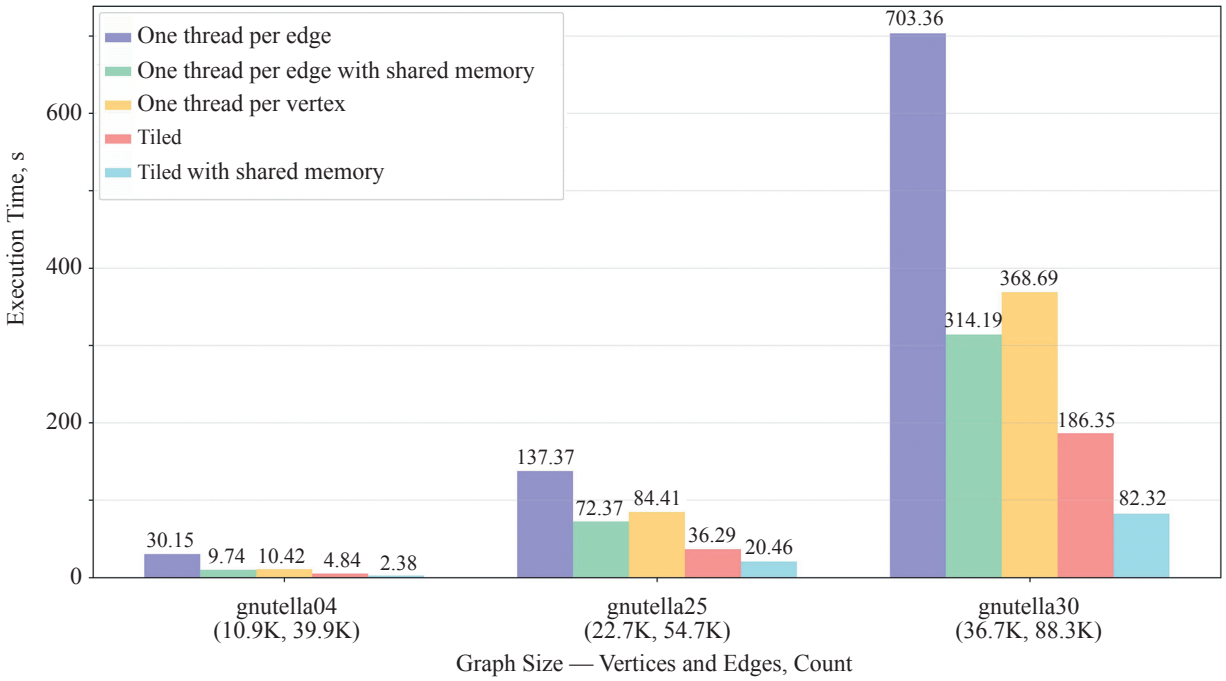

Fig. 2. Execution time comparison for Floyd-Warshall on Stanford's Peer-to-peer network dataset

patterns and balanced workloads yielding the best results [1, 2, 21, 22][1].

A common pattern is that all GPU implementations, even on the largest datasets, outperform the corresponding serial CPU versions on the smallest datasets used for experimenting. This demonstrates the computational advantage that GPU parallelization provides to enable the processing of much larger problem instances in less time than traditional approaches require for smaller problems.

Future Work

Several directions emerge for extending this research, including investigating advanced graph representations, such as ELL format to address control divergence issues [7], developing multi-GPU implementations for very large graphs [8], and adapting algorithms for dynamic graphs where edges are modified during computation. Additional opportunities include implementing memory-efficient techniques like staged loading for graphs exceeding GPU capacity, exploring hybrid CPU-GPU approaches to optimize resource utilization, and developing application-specific optimizations tailored to domains such as transportation or social networks. Comparative analysis with alternative parallel platforms such as OpenCL or distributed computing frameworks would provide broader insights into parallel shortest path computation trade-offs across different architectures.

Conclusion

This paper presented an evaluation of GPU implementations for three fundamental shortest path algorithms: Bellman-Ford, Dijkstra, and Floyd-Warshall using Compute Unified Device Architecture. Through systematic implementation and optimization of multiple variants, we demonstrated significant performance benefits of GPU acceleration for graph processing. Floyd-Warshall achieved the most dramatic improvements with the tiled shared memory implementation delivering approximately 8 times speedup, while Bellman-Ford showed substantial acceleration through flag-based optimization achieving 2.8 times performance improvement. Although Dijkstra algorithm exhibited more limited parallelization benefits due to its sequential nature, it still provided meaningful acceleration over CPU implementations. The results reveal that GPU implementations on large datasets consistently outperform CPU versions on small datasets, highlighting the transformative potential of parallel computing for shortest path problems and enabling practical processing of real-world graph sizes previously computationally prohibitive.

[1] CUDA C++ Programming Guide. Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/, free. English lang. (accessed: 02.06.2025).

References

1. Harish P., Narayanan P.J. Accelerating large graph algorithms on the GPU using CUDA. *Lecture Notes in Computer Science*, 2007, vol. 4873, pp. 197–208. https://doi.org/10.1007/978-3-540-77220-0_21
2. Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. *Introduction to Algorithms*. MIT press, 2009. 1292 p.
3. Katz G.J., Kider J.T. All-pairs shortest-paths for large graphs on the GPU. *Proc. of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 2008, pp. 47–55.
4. Lebedev S.S., Novikov F.A. The Necessary and sufficient condition for Dijkstra's algorithm applicability. *Computer Tools in Education*, 2017, no. 4, pp. 5–13. (in Russian)
5. Lund B., Smith J.W. A multi-stage cuda kernel for floyd-warshall. *arXiv*, 2010, arXiv:1001.4108. https://doi.org/10.48550/arXiv.1001.4108
6. Winkler D., Meister M., Rezavand M., Rauch W. gpuSPHASE—A shared memory caching implementation for 2D SPH using CUDA. *Computer Physics Communications*, 2017, vol. 213, pp. 165–180. https://doi.org/10.1016/j.cpc.2016.11.011
7. Harris M. CUDA *Pro Tip: write flexible kernels with grid-stride loops*. Available at: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops. (accessed: 30.05.2025)
8. Yang S., Liu X., Wang Y., He X., Tan G. Fast All-Pairs Shortest Paths algorithm in large sparse graph. *Proc. of the 37th International Conference on Supercomputing*, 2023, pp. 277–288. https://doi.org/10.1145/3577193.3593728
9. Tang W., Chen T., Armstrong M.P. GPU-accelerated parallel all-pair shortest path routing within stochastic road networks. *International Journal of Geographical Information Science*, 2025, vol. 39, no. 1, pp. 53–85. https://doi.org/10.1080/13658816.2024.2394651
10. Spridon D.E., Deaconu A.M., Tayyebi J. Novel GPU-based method for the generalized maximum flow problem. *Computation*, 2025, vol. 13, no. 2, pp. 40. https://doi.org/10.3390/computation13020040
11. Agarwal P., Dutta M. New approach of Bellman Ford algorithm on GPU using compute unified design architecture (CUDA). *International Journal of Computer Applications*, 2015, vol. 110, no. 13, pp. 1–5. https://doi.org/10.5120/19375-1027
12. Song B. High-performance parallelization of Dijkstra's algorithm using MPI and CUDA. *arXiv*, 2025, arXiv:2504.03667. https://doi.org/10.48550/arXiv.2504.03667
13. Bengtsson M., Wittsten J., Waidringer J. Warehouse storage and retrieval optimization via clustering, dynamic systems modeling, and GPU-accelerated routing. *arXiv*, 2025, arXiv:2504.20655. https://doi.org/10.48550/arXiv.2504.20655
14. Kumar H.S., Singh A., Ojha M.K. Artificial intelligence based navigation in quasi structured environment. *arXiv*, 2024, arXiv:2407.17508. https://doi.org/10.48550/arXiv.2407.17508
15. Morgan N., Yenusah C., Diaz A., Dunning D., Moore J., Heilman E., et al. On a simplified approach to achieve parallel performance and portability across CPU and GPU architectures. *Information*, 2024, vol. 15, no. 11, pp. 673. https://doi.org/10.3390/info15110673
16. Leskovec J., Krevl A. *SNAP Datasets: Stanford large network dataset collection*. 2014. Available at: http://snap.stanford.edu/data
17. Buluç A., Gilbert J.R., Budak C. Solving path problems on the GPU. *Parallel Computing*, 2010, vol. 36, no. 5-6, pp. 241–253. https://doi.org/10.1016/j.parco.2009.12.002
18. Merrill D., Garland M., Grimshaw A. Scalable GPU graph traversal. *ACM SIGPLAN Notices*, 2012, vol. 47, no. 8, pp. 117–128. https://doi.org/10.1145/2370036.2145832
19. Kirk D.B., Hwu W.M.W. *Programming Massively Parallel Processors: a Hands-on Approach*. Morgan Kaufmann, 2016, 576 p.
20. Nickolls J., Buck I., Garland M., Skadron K. Scalable parallel programming with CUDA. *Queue*, 2008, vol. 6, no. 2, pp. 40–53. https://doi.org/10.1145/1365490.1365500
21. Bodra D., Khairnar S. Comparative performance analysis of modern NoSQL data technologies: Redis, Aerospike, and Dragonfly. *Journal of Research, Innovation and Technologies*, 2025, vol. 4, no. 2, pp. 193–200. https://doi.org/10.57017/jorit.v4.2(8).05
22. Khairnar S., Bodra D. Recommendation engine for Amazon magazine subscriptions. *International Journal of Advanced Computer Science and Applications*, 2025, vol. 16, no. 7, pp. 1–8. https://doi.org/10.14569/ijacsa.2025.0160796

Литература

1. Harish P., Narayanan P.J. Accelerating large graph algorithms on the GPU using CUDA // Lecture Notes in Computer Science. 2007. V. 4873. P. 197–208. https://doi.org/10.1007/978-3-540-77220-0_21
2. Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. Introduction to Algorithms. MIT press, 2009. 1292 p.
3. Katz G.J., Kider J.T. All-pairs shortest-paths for large graphs on the GPU // Proc. of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware. 2008. P. 47–55.
4. Лебедев С.С., Новиков Ф.А. Необходимое и достаточное условие применимости алгоритма Дейкстры // Компьютерные инструменты в образовании. 2017. № 4. С. 5–13.
5. Lund B., Smith J.W. A multi-stage cuda kernel for floyd-warshall // arXiv. 2010. arXiv:1001.4108. https://doi.org/10.48550/arXiv.1001.4108
6. Winkler D., Meister M., Rezavand M., Rauch W. gpuSPHASE—A shared memory caching implementation for 2D SPH using CUDA // Computer Physics Communications. 2017. V. 213. P. 165–180. https://doi.org/10.1016/j.cpc.2016.11.011
7. Harris M. CUDA Pro Tip: write flexible kernels with grid-stride loops. URL: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops. (accessed: 30.05.2025)
8. Yang S., Liu X., Wang Y., He X., Tan G. Fast All-Pairs Shortest Paths algorithm in large sparse graph // Proc. of the 37th International Conference on Supercomputing. 2023. P. 277–288. https://doi.org/10.1145/3577193.3593728
9. Tang W., Chen T., Armstrong M.P. GPU-accelerated parallel all-pair shortest path routing within stochastic road networks // International Journal of Geographical Information Science. 2025. V. 39. N 1. P. 53–85. https://doi.org/10.1080/13658816.2024.2394651
10. Spridon D.E., Deaconu A.M., Tayyebi J. Novel GPU-based method for the generalized maximum flow problem // Computation. 2025. V. 13. N 2. P. 40. https://doi.org/10.3390/computation13020040
11. Agarwal P., Dutta M. New approach of Bellman Ford algorithm on GPU using compute unified design architecture (CUDA) // International Journal of Computer Applications. 2015. V. 110. N 13. P. 1–5. https://doi.org/10.5120/19375-1027
12. Song B. High-performance parallelization of Dijkstra's algorithm using MPI and CUDA // arXiv. 2025. arXiv:2504.03667. https://doi.org/10.48550/arXiv.2504.03667
13. Bengtsson M., Wittsten J., Waidringer J. Warehouse storage and retrieval optimization via clustering, dynamic systems modeling, and GPU-accelerated routing // arXiv. 2025. arXiv:2504.20655. https://doi.org/10.48550/arXiv.2504.20655
14. Kumar H.S., Singh A., Ojha M.K. Artificial intelligence based navigation in quasi structured environment // arXiv. 2024. arXiv:2407.17508. https://doi.org/10.48550/arXiv.2407.17508
15. Morgan N., Yenusah C., Diaz A., Dunning D., Moore J., Heilman E., et al. On a simplified approach to achieve parallel performance and portability across CPU and GPU architectures // Information. 2024. V. 15. N 11. P. 673. https://doi.org/10.3390/info15110673
16. Leskovec J., Krevl A. SNAP Datasets: Stanford large network dataset collection. 2014. URL: http://snap.stanford.edu/data
17. Buluç A., Gilbert J.R., Budak C. Solving path problems on the GPU // Parallel Computing. 2010. V.36. N 5-6. P. 241–253. https://doi.org/10.1016/j.parco.2009.12.002
18. Merrill D., Garland M., Grimshaw A. Scalable GPU graph traversal // ACM SIGPLAN Notices. 2012. V. 47. N 8. P. 117–128. https://doi.org/10.1145/2370036.2145832
19. Kirk D.B., Hwu W.M.W. Programming Massively Parallel Processors: a Hands-on Approach. Morgan Kaufmann, 2016. 576 p.
20. Nickolls J., Buck I., Garland M., Skadron K. Scalable parallel programming with CUDA // Queue. 2008. V. 6. N 2. P. 40–53. https://doi.org/10.1145/1365490.1365500
21. Bodra D., Khairnar S. Comparative performance analysis of modern NoSQL data technologies: Redis, Aerospike, and Dragonfly // Journal of Research, Innovation and Technologies. 2025. V. 4. N 2. P. 193–200. https://doi.org/10.57017/jorit.v4.2(8).05
22. Khairnar S., Bodra D. Recommendation engine for Amazon magazine subscriptions // International Journal of Advanced Computer Science and Applications. 2025. V. 16. N 7. P. 1–8. https://doi.org/10.14569/ijacsa.2025.0160796

Authors

Deep Bodra — Magister, Student, Harrisburg University of Science and Technology, Harrisburg,17101, USA, SC 57216618940, https://orcid.org/0009-0009-4173-2447, Deepbodra97@gmail.com
Sushil Khairnar — Magister, Student, Virginia Tech, Virginia, 24061, USA, SC 57204777066, https://orcid.org/0009-0006-5192-0175, sushilk@vt.edu

Авторы

Бодра Дип — магистр, студент, Гаррисбергский университет науки и технологий, Гаррисберг, 17101, США, SC 57216618940, https://orcid.org/0009-0009-4173-2447, Deepbodra97@gmail.com
Хайрнар Сушил — магистр, студент, Технологический институт Вирджинии, Вирджиния, 24061, США, SC 57204777066, https://orcid.org/0009-0006-5192-0175, sushilk@vt.edu

Received 20.06.2025
Approved after reviewing 27.08.2025
Accepted 22.09.2025

Статья поступила в редакцию 20.06.2025
Одобрена после рецензирования 27.08.2025
Принята к печати 22.09.2025