МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2026
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2026
Proceedings of the Fifty-Fifth Spring Conference
of the Union of Bulgarian Mathematicians
Tryavna, Bulgaria, April 5–9, 2026

PERFORMANCE OPTIMIZATION OF HYBRID GNN–LLM PIPELINES FOR CODE COMPARISON

Daniel Damyanov
Department of Information Technologies, St. Cyril and St. Metodius University of Veliko Tarnovo, Bulgaria
e-mail: d.damyanov@ts.uni-vt.bg

This study proposes a set of techniques for optimizing the performance of hybrid pipelines combining large language models (LLMs) and graph neural networks (GNN) for comparison of program code fragments. On the basis on previous work [1] on semantic normalization and graph embeddings, the main computational bottlenecks are identified and solutions are proposed including parallel processing, dynamic skipping of redundant operations, caching, and optimization of graph representation. The article contributes to the practical implementation of semantic-based systems for comparing code in real conditions.

Keywords: performance optimization, GNN, LLM, code comparison, parallel processing, caching, semantic normalization

ОПТИМИЗАЦИЯ НА ПРОИЗВОДИТЕЛНОСТТА НА ХИБРИДНИ GNN-LLM ИЗЧИСЛИТЕЛНИ КОНВЕЙЕРИ ЗА СРАВНЕНИЕ НА КОД

Даниел Дамянов
Катедра „Информационни технологии“, ВТУ „Св. Св. Кирил и Методий“, Велико Търново
e-mail: d.damyanov@ts.uni-vt.bg

Настоящото изследване предлага набор от техники за оптимизиране на производителността на хибридни конвейери, комбиниращи големи езикови модели (LLMs) и графови невронни мрежи (GNN) за сравнение на фрагменти от програмен код. Въз основа на предишна работа [1] по семантична нормализация и графови вграждания, където основните проблемни места бяха забелязани в изчислителния процес и предлагат решения за паралелна обработка, динамично пропускане на излишни операции, кеширане и оптимизация на графовото представяне. Статията допринася за

https://doi.org/10.55630/mem.2026.55.098-106
2020 Mathematics Subject Classification: 68R10, 68U15.

практическата реализация на семантични системи за сравнение на код при реални условия.

Ключови думи: оптимизация на производителността, GNN, LLM, сравнение на код, паралелна обработка, кеширане, семантична нормализация.

1 Introduction

Issues related to performance and code optimization have always concerned researchers and developers [2]. After the successful development of a hybrid architecture for semantic code comparison based on LLM and GNN in previous related work [1] the possible weaknesses of the architecture and algorithms have been identified by the need for its practical optimization. The original pipeline includes several computationally intensive steps: semantic normalization via LLM, generation of multiple columns (AST, CFG, DFG), embedding via GNN and transformer, as well as merge and classification. Despite the high accuracy achieved, the temporal complexity of the system limits its applicability in real environments, especially when processing large codebases or in automatic evaluation systems with strict time constraints. The purpose of this article is to propose and evaluate performance optimization strategies that maintain semantic accuracy but significantly reduce turnaround time. The focus is on parallelization, dynamic strategies for skipping redundant computations, caching intermediate results, and graph performance optimization.

2 Related work

Real-time optimization of ML-pipelines [10], especially those involving GNN upgrades and transformers [8], represent a serious area of study. Several essential directions can be defined in optimizing this problem, consisting in parallel processing of graphs that are huge in volume[3]: dynamic sparsity approach in neural networks, embedding caching and graph representation optimization. All approaches use parameterization, which comes at the expense of additional memory and computational effort during the training and inference of models [5]. Looking at the optimizations that can be made using parallel processing, an effective optimization technique combines parallel processing with sparse matrix representations, enabling efficient graph computations on a single GPU can achieve significant acceleration (compared to CPU and previous GPU approaches) [11]. Speaking of sparsity, it can be divided into two categories: static sparse learning and dynamically sparse learning. Static sparse learning determines the structure of the sparse network at the initial stage of training by using certain preset sparse ratios by layers [3]. Dynamic sparse learning, in turn, is designed to reduce computations as well as memory footprint throughout the training phase [12]. Another widely adopted optimization is embedding caching approach for handling common code fragments reducing computational overhead in which pre-computed vector representations are stored in memory or a database for quick access on subsequent queries.

3 The approach

3.1 Refinement of Amdahl's Law and Use of Hybrid Architectures [6]

Amdahl's Law (1) provides an upper bound on the acceleration achievable through parallelization for a single parallel portion of a program:

$$S = \frac{1}{((1-p) + \frac{p}{n})} \tag{1}$$

where p denotes the fraction of the program that can be parallelized, and n is the number of processors;

In the context of machine-learning systems, where different processing stages can be parallelized to different degrees and on different hardware- Amdahl's law can be generalized through hierarchical parallelization. For our particular pipeline, we can extend it by taking into account hierarchical parallelization. This is a generalized formula for speedup in parallel computing – an extended version of Amdahl's Law when there are several parallelizable parts, each with a different number of processors (2):

$$S_{total} = \frac{1}{(1 - \sum P_i) + \sum(\frac{P_i}{N_i})} \tag{2}$$

where:

S_{total} is the overall acceleration of the program;

P_i is the proportion (percentage) of the program that can be parallelised in Part I;

N_i is the number of processors/cores used for Part I;

$1 - \sum P_i$ is the serial (non-parallelizable) part of the program;

$\sum(\frac{P_i}{N_i})$ is the effective time of parallel parts when executed on the processor N_i;

whereas $\sum P_i \leq 1$.

The original process is strictly consistent, but in the current development, parallelization is proposed, which is carried out by the following stages: independent processing of code fragments within a single comparison task, then parallel generation of AST, CFG and DFG, after normalization, simultaneous execution of GNN and Transformer encoders on the already prepared columns and text. For the original sequential pipeline, the following values for P shall be defined:

$P_1 = 0.30$ (code fragment processing)

$P_2 = 0.5$ (generation of AST, CFG, DFG)

$P_3 = 0.15$ (GNN + Transformer Coding)

$P_4 = 0.05$ (non-parallelizable parts: serialization, IO)

The total parallelizable fraction is therefore $P = 0.95$, while the strictly sequential part accounts for 5% of the execution time. At maximum parallelism ($N_1 = \infty$, $N_2 = 3$, $N_3 = 2$), the experiments were conducted on a Lenovo IdeaPad 5 Pro 16ACH6 laptop, with an AMD Ryzen 5 5600H processor (6 physical cores, 12 logical threads) and 16 GB of RAM. The parallel execution parameters are configured relative to the number of physical CPU cores. Access to the external LLM service is restricted by a semaphore with

capacity C_{LLM}=2, which limits the maximum number of simultaneous LLM requests. It is important to note that **only the code fragment processing stage (P_1) is constrained by the external LLM service**, while the remaining parallel stages are CPU-bound. Consequently, the LLM semaphore does not represent a global bottleneck for the entire pipeline. Under these assumptions, the maximum theoretical acceleration according to the extended Amdahl's law can be calculated as follows:

$$S_{total} = \frac{1}{(1 - 0.95) + (\frac{0.3}{2} + \frac{0.5}{6} + \frac{0.15}{6})} = \frac{1}{0.308} = 3.25 \tag{3}$$

Current result represents **a maximum theoretical acceleration of 3.25x** through parallelization alone. Unlike Amdahl, the Gustafson & Barsis law addresses **weak scaling [7]**, where the problem grows with available resources: $S_{scaled} = N - s * (N - 1)$, where N is the number of processors, s is the sequential part.

In the following lines direct optimizations are given that can improve and optimize calculations in a product environment:

Dynamic skipping of unnecessary normalizations

Not all code snippets need full LLM-normalization. Current system approach applies lexical analysis for complexity (e.g. number of unique identifiers, presence of commutative operations). If the evaluated complexity is below a certain threshold, normalization is skipped and the original columns are used directly.

Caching intermediate results

In order to reduce computational costs and improve the overall performance of the system, part of the intermediate results are cached in RAM or in a local database. Caching is particularly effective for operations with high computational complexity and deterministic outcome. The following intermediate artifacts are cached:

- **Normalized code**, since the normalization process is an expensive operation in terms of time and resources;
- **Generated graph representation**s associated with a hash of the normalized code, allowing reuse with identical input data;
- Graph and text embedding used in the subsequent stages of learning and similarity extraction;
- Optimize the size of columns:
  - Some nodes from the abstract syntactic tree (AST) that do not carry any essential semantic information for the analysis (e.g., certain literals or syntactic constructions without affecting the logic of the program);
  - Using compact node and edge encodings in the combined graph, which reduces memory requirements and allows for faster processing by GNN;
  - Selection of an informative subgraph for GNN embedding based on attention mechanisms that focus on critical areas of the program, such as regions around the main control structures.

3.2 Hierarchical parallelization of stages

Code fragments are processed independently enabling parallel execution of multiple pipeline stages. Access to LLM-based normalization is regulated via a semaphore to limit concurrency. This design minimizes the serial component, in accordance with the extended Amdahl's law (Figure 1).

```
public async Task<CodePairResult[]> ProcessBatchAsync(
        CodePair[] pairs,
        CancellationToken cancellationToken)
    {
        var results = new CodePairResult[pairs.Length];

        await Parallel.ForAsync(
            0,
            pairs.Length,
            new ParallelOptions
            {
                MaxDegreeOfParallelism = _maxDegreeOfParallelism,
                CancellationToken = cancellationToken
            },
            async (i, ct) =>
            {
                await _llmSemaphore.WaitAsync(ct);
                CodePair normalized;
                try
                {
                    normalized = await LLMNormalizer.NormalizeAsync(pairs[i], ct);
                }
                finally
                {
                    _llmSemaphore.Release();
                }
                results[i] = await ProcessSinglePairAsync(normalized, ct);
            });

        return results;
    }
```

Figure 1. Process batch implementation C#

In the context of the ParallelCodeProcessor, and with reference to the explanation of the implementation (Formula 2), the pipeline can be designed as a hierarchically parallelized system composed of several successive stages summarized in Table 1. Each stage exhibits a different degree of parallelization and may utilize a different number of processing units allowing for a more accurate estimation of the overall speedup:

Table 1: Stages of parallel implementation

Stage i	**Description**	**Parallelizable partition** P_i	**Number of resources** N_i
1	LLM Normalization	P_1	$N_1 = C_{LLM}$
2	Graph processing embedding	P_2	N_2 =CPU/GPU cores
3	Post-processing	P_3	N_3 =MaxDegreeOfParallelism

The Serial Part $1 - \sum P_i$ includes initialization, synchronization, serialized requests to external services. Therefore, the maximum acceleration of the pipeline is limited by the size of the semaphore for LLM requests and by the least parallelizable stage. Even with a high degree of parallelism at the other stages, the limit on the number of competing LLM applications constitutes a dominant serial component that sets the upper limit of

the S_{total}.

Rate limiting + exponential backoff (production-grade)

When interacting with external LLM services that impose restrictions on the frequency and number of requests, it is necessary to implement mechanisms for load control and resilience in case of temporary failures. The main goals in the use of *rate limiting and exponential backoff* are to prevent exceeding the API limits imposed by the LLM service provider (rate limits, quota), Figure 2, to ensure the stability of the system in case of temporary errors, related to congestion or network latency, controlling competitive access to limited external resources through a defined maximum number of concurrent requests. It also achieves a reduction in the probability of cascading failures in the parallel pipeline under high load, preserving the theoretical acceleration defined by Amdahl's extended law by minimizing the uncontrollable serial component. In the pipeline under consideration, rate limiting is implemented through a semaphore that limits the number of competing LLM requests (C_{LLM}), while Exponential Backoff applies when errors occur related to exceeding the limits or temporarily unavailability of the service. This combination allows for adaptive load management without compromising on the parallelization of the other stages.

```
private async Task<T> ExecuteWithBackoffAsync<T>(
    Func<CancellationToken, Task<T>> action,
    CancellationToken ct,
    int maxRetries = 5)
{
    var delay = TimeSpan.FromSeconds(1);

    for (int attempt = 0; attempt < maxRetries; attempt++)
    {
        try
        {
            return await action(ct);
        }
        catch (RateLimitException) when (attempt < maxRetries - 1)
        {
            await Task.Delay(delay, ct);
            delay = delay * 2; // exponential backoff
        }
    }
    throw new InvalidOperationException("LLM request failed after retries.");
}
```

Figure 2. Exponential backoff retry mechanism

Resource Profile:

1. Using CPU Cores and ThreadPool for Load Balancing
2. GPU memory for batch processing of ML operations
3. Network bandwidth to limit competing LLM API requests
4. Parallel graph generation (AST, CFG, DFG)

Graph operations demonstrate different characteristics of parallelism, which are summarized in Table 2:

Table 2: Summary analysis of the types of graphs and their characteristics

Graph type	Parallelism	Complexity	Memory	Dependencies
AST	Inner tree	$O(n)$	Low	Minimal
CFG	Over basic blocks	$O(n)$	Medium	Control dependencies
DFG	Over data flow ways	$O(n^2)$	High	Data dependencies

3.3 Experimental evaluation

The following hardware, software resources and analysis metrics were used to perform the tests and subsequent analysis:

Hardware: AMD Ryzen 5 5600H (6C / 12T), 16 GB RAM, integrated GPU, outer GPU NVIDIA RTX 3050 4GB GDDR6

Software: C#, Python (PyTorch, Hugging Face Transformers), Graph-tool.

Data: Extended dataset of 2000 triplets of code fragments, including samples of different levels of complexity.

Metrics: Time to analyze a pair of code fragments, F1-Score, Precision, Recall, memory load.

Summary results and analysis

The proposed optimizations unequivocally lead to a significant reduction in analysis time (over 58%), Table 3, with minimal loss of accuracy ($\Delta F1 \leq 0.004$). Dynamic skipping reduces calls to LLMs by about 35%. Caching shows the greatest effect on repetitive code patterns. The theoretical model based on Amdahl's extended law predicts maximum acceleration $S_{total} = 3.25$ for the evaluated hardware configuration. Working with real application the observed speedup reaches approximately $2.38\times$, approaching to about 73% of the theoretical upper bound. This discrepancy is primarily attributed to unavoidable practical factors, including synchronization overhead, partial serialization within the LLM-constrained stages, and latency introduced by external service requests. The fact that the empirically measured acceleration remains below S_{total} confirms that the theoretical value represents a correctly formulated upper limit, rather than an expected practical outcome.

Table 3: Results

Configuration	Medium time (ms)	F1-Score	Time reduction
Original pipeline	298	0.917	–
Parallelization	210	0.917	29.5%
Dynamic skipping	185	0.915	37.9%
Caching	155	0.917	48.0%
Graph optimization	142	0.914	52.3%
All optimizations	125	0.913	58.1%

4 Discussion

Generalization across hardware configurations

Although the experimental evaluation was conducted on a specific hardware setup, the proposed optimizations are largely hardware-agnostic. Techniques such as dynamic comparison skipping and embedding caching operate at the algorithmic and architectural level rather than relying on low-level hardware acceleration. As a result, similar performance trends are expected across different CPU-and-memory-based environments, with absolute execution times varying depending on available computational resources.

Interaction effects between optimizations

The applied optimizations are not independent and exhibit complementary interaction effects. In particular, caching of normalized code embeddings amplifies the effectiveness of dynamic skipping, as repeated or structurally similar submissions can be filtered early in the pipeline without invoking expensive LLM or GNN computations. While this interaction significantly reduces reliance on external ML services (e.g., LLM APIs), it introduces moderate deployment complexity and additional memory overhead for cache management. Nevertheless, these trade-offs are acceptable in scenarios such as automatic grading and large-scale code monitoring, where multiple comparisons are executed in parallel and cost efficiency is critical. Optimizations are especially effective in the context of **automatic grading of student tasks** and **code-based monitoring**, where multiple comparisons are performed in parallel. Dynamic skipping and caching reduce dependence on external ML services (LLM API), which reduces costs and increases autonomy. Limitations include slightly increased deployment complexity and the need for additional memory for caching.

5 Conclusion

Applying Discriminative Projection Using Fisher's Linear Discriminant

After obtaining joint graph–text embeddings from the GNN and Transformer encoders, an optional linear discriminative projection based on Fisher's Linear Discriminant (FLD) is applied. The objective is to maximize inter-class variance while minimizing intra-class variance between semantically similar and non-similar code pairs. Unlike deep models, FLD introduces negligible computational overhead and serves as an effective dimensionality reduction and class-separability enhancement technique prior to the final similarity estimation. This step is particularly beneficial in production scenarios with cached embeddings and dynamic skipping, where fast discrimination is required. The transition from the original sample space to a smaller dimensional space is related to the accuracy and separability of the data [9]. Current work and optimizations demonstrate that hybrid GNN–LLM code comparison pipelines could be significantly accelerated through a combination of parallelization, dynamic strategies, caching, and graph optimization. Experiments have shown a reduction in analysis time by over 58% while maintaining high semantic accuracy. This opens up the possibility of wider practical application in real systems. Subsequent future research may include the following improvements:

1. Integrated with distributed computing for multi-large codebases.
2. Adaptive dynamic skipping thresholds trained with reinforcement training.
3. Quantizing and packaging GNN and Transformer models for even faster edge device execution.

References

[1] Damyanov D., Semantic normalization on graph models searching structure matchings in code, Mathematics and Education in Mathematics (to appear).

[2] Donchev, I., An alternative to virtual functions in modern C++. Application in training, Mathematics, Computer Science and Education, Vol. 8, Issue 2, (2025), pp. 144-159
[3] Dong X., Gu Y., Sun Y., Wang L., PASGAL: Parallel And Scalable Graph Algorithm Library, (2024), https://doi.org/10.48550/arXiv.2404.17101
[4] Gale T., Elsen E., Hooker S., The State of Sparsity in Deep Neural Networks, (2019), DOI: 10.48550/arXiv.1902.09574
[5] Hoefler T., Alistarh D., Dryden T., Peste A., Sparsity in Deep Learning: Pruning and growth for efficient, Journal of Machine Learning Research 23, (2021), pp. 1-124, DOI: 10.48550/arXiv.2102.00554
[6] Holm M., Scientific Computing on Hybrid Architectures, Department of Information Technology, Uppsala University, ISSN 1404-5117, (2013), Sweden
[7] Karbowski A., Amdahl's and Gustafson-Barsis laws revisited, Pomiary Automatyka Robotyka, vol. 29(4), (2025), pp. 35-38, https://doi.org/10.48550/arXiv.0809.1177
[8] Pugacheva D., Ermakov A., Lyskov I., Igor M., Zotov Y., Enhancing GNNs Performance on Combinatorial Optimization by Recurrent Feature Update, Cornell University, (2024), DOI: 10.48550/arXiv.2407.16468.
[9] Petrov M., Dochkova J., Comparison of the PCA and FLD Approaches in Glial Tumors Classification Systems, Baltic J. Modern Computing, Vol. 13, No. 1, (2025), pp. 221-232, https://doi.org/10.22364/bjmc.2025.13.1.12
[10] Ramisetty S., Chandrasekaran T., Eruvaram V. K., Pulicharla M. R., Optimizing Real-Time Data Pipelines for Machine Learning: A Comparative Study of Stream Processing Architectures, World Journal of Advanced Research and Reviews, (2024), pp.1653–1660, DOI: 10.30574/wjarr.2024.23.3.2818
[11] Yang C., Buluc A., Owens J., GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU, ACM Transactions on Mathematical Software (TOMS), Volume 48, Issue 1, (2022), pp.1-51, https://doi.org/10.1145/3466795
[12] Yin L., Li G., Fang M., Shen L., Huang T., Wang Z., Menkovski V., Ma X., Pechenizkiy M., Liu S., Dynamic Sparsity Is Channel-Level Sparsity Learner, 37th Conference on Neural Information Processing Systems NeurIPS, (2023), pp. 67993 – 68012, https://doi.org/10.48550/arXiv.2305.19454