

Agent-Guided CUDA Kernel Generation for High-Performance Numerical Computing

Authors

Oliver Bennett^{1*}, James Whitmore², Sophie Harrington³

Affiliations

School of Informatics, University of Edinburgh, Informatics Forum, 10 Crichton Street, Edinburgh EH8 9AB, United Kingdom

*Corresponding author: oliver.bennett@ed.ac.uk

Abstract

High-performance numerical computing requires code that is not only functionally correct but also optimized for parallel execution, memory access, and hardware utilization. Current large language models can generate CUDA kernels, but they often produce inefficient thread mapping, poor memory coalescing, excessive synchronization, and unstable performance across GPU architectures. This study examines agent-guided CUDA kernel generation for numerical computing tasks. We propose KernelAgent, a multi-agent model composed of a mathematical decomposition agent, a CUDA code generation agent, a correctness testing agent, a profiling agent, and a performance repair agent. The mathematical decomposition agent converts numerical tasks into parallelizable operations. The code generation agent produces CUDA C++ kernels and host-side launch code. The correctness testing agent compares GPU outputs with CPU reference implementations under random and boundary inputs. The profiling agent collects occupancy, memory throughput, warp divergence, register usage, and kernel execution time. The performance repair agent rewrites kernels using shared memory tiling, loop unrolling, vectorized memory access, and block-size adjustment. The model was evaluated on 860 numerical programming tasks, including matrix multiplication, convolution, prefix sum, stencil computation, reduction, sparse operations, and element-wise tensor transformations. Experiments generated 10,320 CUDA programs and ran 64,800 correctness and profiling tests on NVIDIA RTX 4090, A100, and L40S GPUs. Compared with a single-agent code generator, KernelAgent improved functional correctness from 62.1% to 80.4%. Average kernel execution time decreased by 36.7%, global memory load efficiency increased from 68.5% to 84.9%, and achieved occupancy increased from 51.3% to 69.8%. For matrix and convolution tasks, KernelAgent achieved a 1.72 $\times$ average speedup over unoptimized generated kernels. These findings indicate that agent-level profiling and repair can substantially improve the quality of generated GPU programs.

Keywords: CUDA code generation; numerical computing; GPU optimization; performance profiling; multi-agent systems; parallel programming; kernel repair

1. Introduction

Graphics processing units (GPUs) have become a fundamental computing platform for scientific computing, numerical simulation, large-scale data analysis, and machine learning because their massively parallel architecture can execute thousands of lightweight threads concurrently. Their computational capability has enabled substantial acceleration in applications ranging from dense linear algebra and physical simulation to deep neural networks and large-scale optimization. However, the theoretical throughput of modern GPUs does not automatically translate into high application performance[1-4]. The efficiency of a CUDA program depends strongly on how the underlying numerical algorithm is mapped to threads, thread blocks, memory hierarchies, and hardware execution units. A kernel may produce numerically correct results while using only a small fraction of the available computational resources because of uncoalesced global-memory accesses, insufficient data reuse, excessive synchronization, thread divergence, high register pressure, poor shared-memory utilization, or low multiprocessor occupancy. These effects are often coupled. Increasing the number of threads may improve parallelism while simultaneously increasing register consumption, whereas aggressive shared-memory tiling may reduce global-memory traffic but restrict the number of resident blocks. CUDA optimization therefore requires coordinated reasoning about numerical structure, data layout, parallel decomposition, memory behavior, and GPU architecture[5-8]. Compiler-based and search-based systems have reduced part of this optimization burden by automatically exploring alternative implementations. Tensor-level intermediate representations and scheduling frameworks allow computational graphs to be transformed through loop reordering, tiling, vectorization, memory placement, and other scheduling operations [1]. Hardware-aware systems have further demonstrated that tile size, memory hierarchy, and data reuse can be jointly optimized according to device characteristics [9-13]. Such methods have produced strong results for dense tensor computations and other structured operators. Automated scheduling, kernel fusion, and domain-specific compilation have also reduced the need for manually written low-level GPU code [4]. Nevertheless, their effectiveness is often constrained by the representation and transformation space defined in advance. A compiler can search efficiently when an operator can be expressed within a known schedule space, but numerical programming contains substantially more structural diversity[14-17]. Matrix multiplication benefits from hierarchical tiling and reuse of shared-memory fragments, reductions require carefully staged aggregation and synchronization, prefix scans contain nontrivial inter-thread dependencies, stencil computations repeatedly access neighboring elements, and sparse kernels are dominated by irregular memory access and workload imbalance. The problem is therefore not limited to selecting a tile size or block dimension. Efficient GPU execution frequently depends on identifying the computational structure of the problem before a suitable kernel organization can be selected. Related research in machine learning has shown that a model may exploit multiple shortcut signals simultaneously and that explicitly identifying and removing such shortcuts can improve the separation between

task-relevant structure and spurious patterns [6]. Although this problem arises in a different computational context, it highlights an important principle for automatic kernel generation: systems that rely mainly on recurring surface patterns may produce solutions that appear effective on familiar tasks while failing to capture the structural properties that determine robust performance across problem classes and hardware settings.

The emergence of large language models (LLMs) for source-code generation creates a different opportunity for GPU programming. General-purpose code models have demonstrated substantial capability in program completion, function synthesis, code translation, debugging, and instruction-guided software development [7,8]. Unlike conventional optimization systems that operate within explicitly constructed transformation spaces, LLMs can generate new control structures, memory layouts, helper functions, and algorithmic implementations directly from natural-language descriptions or reference programs. This flexibility is particularly attractive for CUDA because different numerical problems may require qualitatively different parallelization strategies. An LLM can, in principle, decide whether an operation should be decomposed by rows, columns, tiles, segments, warps, or hierarchical reductions rather than merely tuning parameters inside a predetermined implementation. It can also rewrite the implementation when the original decomposition is unsuitable. However, general code-generation ability does not guarantee effective GPU programming. CUDA kernels must satisfy several constraints simultaneously: the code must compile, numerical outputs must remain within acceptable error tolerances, memory accesses should be efficient, synchronization must be correct, the amount of parallel work should be sufficient, and resource usage must remain compatible with hardware limits. A generated kernel can therefore be syntactically valid and functionally correct while still performing substantially worse than a conventional library implementation [9,10]. Recent research has consequently shifted from single-pass code generation toward iterative generation and feedback-based refinement. Self-debugging methods allow models to inspect execution outcomes and revise generated programs after compilation or runtime failures [11,12]. Multi-agent software-development frameworks distribute planning, coding, testing, and review across specialized model roles, providing additional structure to the generation process [13,14]. Other systems separate code synthesis, test generation, and execution feedback so that correctness can be evaluated repeatedly rather than assumed from the initial response [15,16]. These approaches demonstrate that execution feedback can substantially improve program reliability, but conventional software correctness provides only a partial objective for GPU kernels. CPU-oriented code-generation benchmarks typically regard a program as successful once it compiles and passes functional tests. CUDA optimization requires another level of evaluation because many functionally equivalent kernels exhibit large differences in latency, throughput, memory traffic, occupancy, synchronization cost, and warp efficiency. Moreover, the causes of poor correctness and poor performance are fundamentally different. A compilation failure may result from invalid syntax or an incorrect API call; a numerical mismatch may arise from indexing

or synchronization errors; low performance may instead result from memory serialization, excessive register allocation, poor occupancy, redundant global-memory transactions, or an unsuitable decomposition of the computation. Treating all of these outcomes as a single undifferentiated generation failure can make iterative repair inefficient. Several recent benchmarks and optimization systems have begun to consider GPU-specific performance as an explicit objective. KernelBench evaluates whether generated GPU kernels are not only functionally correct but also faster than reference implementations, providing an important benchmark for performance-aware kernel generation [17,18]. Iterative CUDA-generation systems have further incorporated compiler output, correctness testing, and hardware measurements into repeated optimization loops [19]. Other frameworks separate high-level optimization decisions from low-level kernel implementation, allowing algorithmic strategies and CUDA code generation to be handled at different stages [20-25]. Research on Triton kernel generation has expanded evaluation across multiple task groups and has shown substantial differences in generated-kernel performance depending on workload characteristics [26-30]. Broader kernel-generation benchmarks have also examined the generalization of generated implementations across a wider range of operators and GPU platforms [23,24]. Collectively, these studies indicate that LLM-based kernel generation is moving beyond simple code synthesis toward performance-oriented systems in which generated implementations are evaluated through actual execution. They also reveal an important difficulty: an optimization that performs well for one operator, tensor shape, or GPU architecture may provide little benefit under a different configuration. Performance optimization therefore requires feedback that is sufficiently detailed to distinguish the underlying bottleneck rather than relying only on total execution time. Important gaps remain in the current literature. Existing evaluations are still heavily concentrated on machine-learning operators, dense tensor programs, or Triton-based implementations, whereas scientific and numerical computing contains a broader set of computational structures. Prefix sums, irregular reductions, multidimensional stencils, sparse matrix operations, neighborhood transformations, and mixed compute-memory kernels create substantially different requirements for parallel decomposition and synchronization[31-34]. A framework evaluated mainly on dense operators may therefore overestimate its ability to generalize to numerical programming as a whole. Another limitation is that many systems begin directly with kernel generation from an operator specification or reference implementation. This approach implicitly requires the code-generation model to infer mathematical structure, determine parallel dependencies, select a mapping strategy, write low-level CUDA code[35-38], and optimize hardware behavior within the same reasoning process. When a generated implementation performs poorly, it becomes difficult to determine whether the failure originates from an incorrect interpretation of the numerical algorithm or from an inefficient CUDA implementation of an otherwise suitable decomposition. The feedback structure used during optimization also remains relatively coarse in many systems. Compilation status, test results, and execution time are useful, but they do not fully describe why a

kernel is slow. Two kernels with similar runtime can be limited by entirely different factors[39-41]. One may suffer from low achieved occupancy caused by excessive register use, another from low memory throughput caused by non-coalesced accesses, and another from severe branch divergence or excessive synchronization. Optimization based only on runtime can lead to repeated modifications without a clear connection between the observed bottleneck and the proposed repair. Separating correctness validation from performance diagnosis provides a more interpretable optimization process because a kernel can be functionally stabilized before architectural properties are modified. Hardware portability creates an additional challenge. GPUs such as the NVIDIA RTX 4090, A100, and L40S differ in memory systems, compute capability, cache behavior, register resources, and workload characteristics. A kernel tuned aggressively for one architecture may therefore fail to preserve its advantage on another device. Evaluating only a single GPU provides limited evidence regarding whether an automatically generated optimization reflects a general computational improvement or a device-specific configuration.

This study proposes KernelAgent, a multi-agent framework for CUDA kernel generation and performance optimization that explicitly separates mathematical reasoning, code construction, correctness validation, hardware profiling, and performance repair. The framework begins with a mathematical decomposition agent that analyzes the numerical specification and converts it into parallel computational units, including the required data dependencies, reduction structure, neighborhood relationships, and opportunities for reuse. This decomposition is passed to a CUDA code-generation agent, which constructs an executable CUDA C++ implementation according to the identified parallel structure. A dedicated correctness-testing agent then compiles and executes the generated program and compares its outputs with CPU reference implementations under multiple input configurations. Performance optimization is initiated only after functional correctness has been established. The profiling agent collects hardware-level measurements including kernel execution time, occupancy, memory throughput, warp divergence, register consumption, and other resource indicators. These measurements are provided to a performance repair agent that identifies likely bottlenecks and generates targeted modifications to thread organization, memory-access patterns, synchronization, data reuse, or kernel structure. This separation is intended to prevent numerical repair and performance tuning from interfering with each other and to make the reason for each optimization step more explicit. The study evaluates KernelAgent on 860 numerical programming tasks spanning matrix multiplication, convolution, prefix sum, stencil computation, reduction, sparse operations, and element-wise transformations. The evaluation produces 10,320 CUDA programs and performs 64,800 correctness and profiling tests across NVIDIA RTX 4090, A100, and L40S GPUs. This experimental design is intended to examine several related questions within a common framework: whether explicit mathematical decomposition improves the reliability of generated CUDA implementations; whether specialized agent roles increase functional correctness relative to direct generation; whether hardware profiling provides actionable information for iterative optimization; whether performance

improvements are preserved across heterogeneous numerical workloads; and whether generated optimizations remain effective when transferred among different GPU architectures. By combining task-level mathematical reasoning with hardware-level execution feedback, the proposed framework aims to bridge the gap between general-purpose LLM code generation and architecture-aware GPU optimization. The broader significance of this work lies in reducing the amount of expert intervention required to transform numerical algorithms into efficient CUDA implementations while preserving interpretability of the optimization process. Rather than treating CUDA generation as a single code-synthesis problem, KernelAgent formulates it as a sequence of mathematical analysis, implementation, verification, diagnosis, and repair, providing a systematic foundation for automated high-performance numerical programming on modern GPUs.

2. Materials and Methods

2.1 Task Samples and Computing Environment

The study included 860 numerical programming tasks from seven common GPU workload types: matrix multiplication, convolution, prefix sum, stencil computation, reduction, sparse operations, and element-wise tensor operations. These tasks covered different computing patterns, such as dense calculation, local data access, stepwise aggregation, irregular indexing, and memory-bound processing. Each task contained a mathematical description, input and output rules, valid data ranges, and a CPU reference program. Several input sizes were used for each task type to avoid testing only one problem scale. All CUDA programs were compiled and tested on NVIDIA RTX 4090, A100, and L40S GPUs. The CUDA version, compiler settings, numerical precision, and test inputs were kept the same for methods compared on the same GPU. A total of 10,320 CUDA programs were generated from the 860 tasks.

2.2 Experimental Design and Control Settings

KernelAgent was compared with a single-agent CUDA generator under the same task descriptions, generation limits, compiler settings, and hardware conditions. The single-agent method generated CUDA code directly from each numerical task and did not use separate mathematical decomposition, profiling, or performance repair. KernelAgent used five agents for mathematical decomposition, CUDA code generation, correctness testing, performance profiling, and code repair. Ablation tests were also carried out by removing mathematical decomposition, profiler feedback, or performance repair from the full system. The same test inputs were used for the experimental and control groups. Runtime comparison was performed only when both programs passed the required correctness tests. This rule prevented incorrect but fast kernels from being counted as successful cases.

2.3 Measurement Methods and Quality Control

Correctness was measured by comparing GPU outputs with the corresponding CPU reference results. Each program was tested with random inputs, boundary values, and different input sizes. Floating-point outputs were accepted only when their absolute and relative errors were within the preset limits. Compilation errors, illegal memory access, timeout, and output mismatch were recorded as failures. Programs that passed the correctness tests were then measured for kernel execution time, achieved occupancy, global memory load efficiency, memory throughput, warp divergence, and register use. CUDA event timing and NVIDIA profiling tools were used for these measurements. Warm-up runs were performed before timing to reduce startup effects. Each valid kernel was then executed several times under the same settings, and the median runtime was used as the final value. All compared programs were compiled with the same optimization level and tested on the same GPU. In total, 64,800 correctness and profiling tests were completed.

2.4 Data Processing and Performance Metrics

The experimental records were grouped by task type, generation method, and GPU model. Functional correctness was measured as the proportion of generated programs that passed all required tests. The correctness rate was calculated as

$$CR = \frac{N_p}{N} \times 100\%$$

where N_p is the number of programs that passed all correctness tests and N is the total number of generated programs. Kernel speedup was calculated only for program pairs that produced correct results. For task i , the speedup was defined as

$$S_i = \frac{T_{i,control}}{T_{i,optimized}}$$

where $T_{i,control}$ is the execution time of the control kernel and $T_{i,optimized}$ is the execution time of the optimized kernel. A value of $S_i > 1$ indicates that the optimized kernel runs faster than the control kernel. Execution time, achieved occupancy, global memory load efficiency, warp divergence, and register use were summarized by task type and GPU model. Relative changes were calculated against the single-agent baseline to allow comparison across different GPU platforms.

2.5 Statistical Analysis and Reproducibility

Statistical analysis was performed at both the individual-task level and workload-category level. KernelAgent and the control method were tested with the same tasks, input sizes, and GPU devices, so paired comparisons were used. Mean, median, standard deviation, and interquartile range were calculated for execution time and hardware measurements. The distribution of paired differences was checked before significance testing. A paired parametric test was used when the data followed an approximately normal distribution, while a paired non-parametric test was used otherwise. Statistical significance was defined as $p < 0.05$. Results were also analyzed separately

for RTX 4090, A100, and L40S GPUs to check whether the performance changes were consistent across different GPU architectures. Each task, generated program, correctness result, profiling record, and hardware run was assigned a unique experiment ID so that the reported results could be traced to their original programs and test settings.

3. Results and Discussion

3.1 Functional Correctness of Generated CUDA Programs

KernelAgent achieved a functional correctness rate of 80.4%, compared with 62.1% for the single-agent generator. This represents an increase of 18.3 percentage points. The failure rate decreased from 37.9% to 19.6%. The result shows that separating mathematical decomposition, code generation, and correctness testing can reduce errors during CUDA generation. This effect is important for prefix sum, reduction, stencil, and sparse operations, where data dependence and index boundaries often lead to incorrect outputs even when the code compiles successfully. The overall comparison is shown in Fig. 1. Xiong et al. (2026) reported a similar problem in KernelBench, where many generated GPU kernels failed because of compilation errors, runtime errors, or incorrect numerical outputs. Their results also showed that successful execution does not always mean that the generated kernel is correct. The present study extends this finding to a broader set of numerical tasks and shows that a separate correctness-testing stage can remove invalid kernels before performance optimization [25,26].

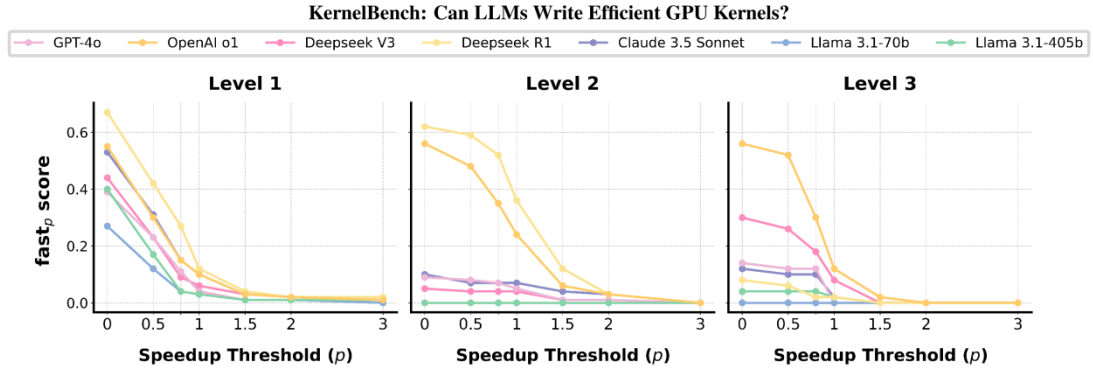


Fig. 1. Comparison of functional correctness and execution time between KernelAgent and the single-agent method.

3.2 Kernel Runtime and Speedup

Among the kernels that passed all correctness tests, KernelAgent reduced average execution time by 36.7% compared with the single-agent baseline. This corresponds to an average speedup of about 1.58×. Matrix multiplication and convolution achieved a higher average speedup of 1.72×. These tasks have regular memory layouts and repeated arithmetic operations. They can therefore benefit more from shared-memory tiling, loop unrolling, and block-size adjustment. Sparse operations and tasks with strong data dependence showed smaller gains because their

performance is more affected by irregular memory access and synchronization. KernelBench also found that only a limited number of one-step generated kernels were faster than strong reference implementations, while repeated execution and revision improved performance [27,28]. CudaForge reported an average speedup of about 1.68 $\times$ after adding hardware feedback to its optimization process. Although the datasets and baselines are different, these studies support the same general result: repeated testing and hardware-based revision can improve the runtime of generated CUDA kernels.

3.3 Memory Access and GPU Utilization

KernelAgent also improved several hardware-level measures. Global memory load efficiency increased from 68.5% to 84.9%, while achieved occupancy increased from 51.3% to 69.8%. The changes are presented in Fig. 2. Higher memory-load efficiency means that a larger part of each memory transaction was used by the kernel, which reduced unnecessary global-memory access. Higher occupancy increased the number of active warps and helped hide memory and instruction latency. These changes provide a direct explanation for part of the runtime reduction. They also show why execution time alone is not enough for kernel repair. Runtime can show whether a kernel is faster, but it cannot identify the source of the improvement or the main bottleneck. CudaForge also uses profiler information, including memory throughput, occupancy, and warp behavior, to guide later kernel changes [29,30]. The results of the present study show that profiler-based repair can improve both memory access and GPU resource use.

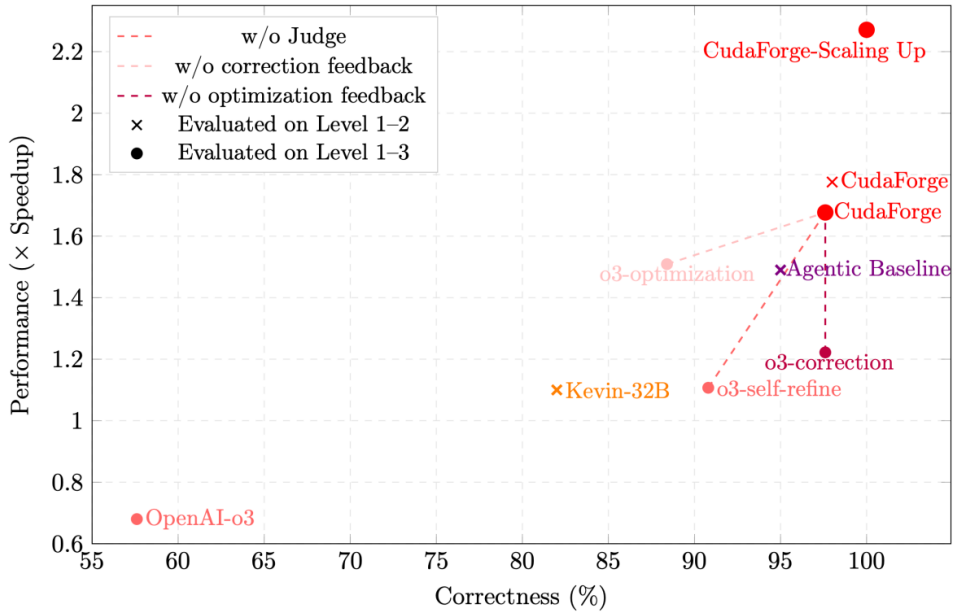


Fig. 2. Comparison of memory load efficiency and GPU occupancy before and after kernel optimization.

3.4 Cross-GPU Performance and Discussion

The 64,800 correctness and profiling tests were carried out on RTX 4090, A100, and L40S GPUs. These devices differ in compute capacity, memory bandwidth, cache design, and available resources. KernelAgent showed improvements across all three GPU models, indicating that the method is not limited to one hardware setting. However, the best kernel configuration was not always the same across devices. Block size, shared-memory use, register use, and memory access can affect runtime differently on different GPU architectures. KernelBench also reported that the relative performance of generated kernels can change across GPU devices. CudaForge reached a similar conclusion when testing kernels on several NVIDIA GPUs and used hardware feedback to adjust the optimization process for each device. These results show that CUDA generation should not rely on one fixed optimization rule. Instead, kernel repair should use measurements from the target device. A limitation of the present study is that all experiments were conducted on NVIDIA GPUs. Further tests on other accelerator platforms are needed to examine whether the same method can be transferred to different hardware systems.

4. Conclusion

This study proposed KernelAgent, a multi-agent method for CUDA kernel generation and optimization in numerical computing. The method separates mathematical decomposition, code generation, correctness testing, performance profiling, and kernel repair. KernelAgent increased functional correctness from 62.1% to 80.4% and reduced average kernel execution time by 36.7%. Global memory load efficiency increased from 68.5% to 84.9%, and GPU occupancy increased from 51.3% to 69.8%. Matrix multiplication and convolution tasks achieved an average speedup of 1.72 $\times$. These results show that combining correctness testing with hardware feedback can improve both code quality and execution performance. The main contribution of this study is a multi-stage workflow that links numerical task analysis with GPU performance optimization. This method can be used in scientific computing, numerical simulation, machine learning, and other GPU-based applications. It can also reduce the need for repeated manual CUDA tuning. However, the experiments were limited to three NVIDIA GPUs and a fixed set of numerical tasks. Future work should test more complex workloads, larger task sets, and other GPU or accelerator platforms to evaluate the method under a wider range of computing conditions.

References

1. Chen, F., Liang, H., Li, S., Yue, L., & Xu, P. (2025). Design of Domestic Chip Scheduling Architecture for Smart Grid Based on Edge Collaboration.
2. Que, H. H., Jin, Y., Wang, T., Liu, M. K., Yang, X. H., & Qiao, F. (2023). A Survey of Approximate Computing: From Arithmetic Units Design to High-Level Applications. *Journal of Computer Science and Technology*, 38(2), 251-272.

3. Liang, R., Feifan, F. N. U., Liang, Y., & Ye, Z. (2025). Emotion-Aware Interface Adaptation in Mobile Applications Based on Color Psychology and Multimodal User State Recognition. *Frontiers in Artificial Intelligence Research*, 2(1), 51-57.
4. Zhang, Z. (2026). A Study on the Impact of Cost Presentation on Decision-making Bias and Cancellation Behavior in Online Subscriptions. Available at SSRN 7349839.
5. Fan, Y., Miyazaki, T., Tang, Z., Wang, J., Huang, Y., & Omachi, S. (2025, November). Scene Text Reconstructor: A Contextual-Aware Masking Framework for Pre-training Text Detectors. In *International Conference on Neural Information Processing* (pp. 181-195). Singapore: Springer Nature Singapore.
6. Zheng, J., & Makar, M. (2022). Causally motivated multi-shortcut identification and removal. *Advances in Neural Information Processing Systems*, 35, 12800-12812.
7. Qi, C., & Qiao, X. (2026). Building Reliable AI Systems: A Framework That Bridges Architecture and Operations. Available at SSRN 6795298.
8. Xiong, W., Zeng, Y., & Guo, Y. (2026). A Study on the Impact of MEP Space Organization in Large-Scale Commercial Complexes on Investment Efficiency and Its Mechanisms.
9. Mini Han Wang , " AI-Powered Innovations in Ophthalmic Diagnosis and Treatment ", Bentham Science Publishers (2025). <https://doi.org/10.2174/97988988122871250101>
10. Hong, Z., Liang, S., Xu, T., & Chen, H. (2026). A Study on Failover Verification and Recovery Objective Prediction for Cross-Region Cloud Services.
11. Huang, J., Xu, T., Yang, J., & Yin, J. (2026). A Graph Neural Network Approach for Financial Data Validation and Risk Identification in Large-Scale Intercompany Transactions. Available at SSRN 7175058.
12. Zhao, J., Fan, J., & Li, L. (2026). A Study on an Explainable Causal-Enhanced LLM Agent for Predicting the Forming Quality of Automotive Component Materials.
13. Liu, W., Zhao, R., Sha, Z., Cui, Q., & Zhang, Y. (2026). Mapping the City Through the Lens of Language Models. *arXiv preprint arXiv:2608.02971*.
14. You, S. (2026). Verifiable Audit Mechanisms in AI Compliance Automation: Scalability. Available at SSRN 6547458.
15. Jiao, Y., Shi, T., Zhao, B., & Wang, A. (2026). Retrieval-Guided Structured Reasoning and Interpretable Representation Learning for Large-Scale Video–Language Models.
16. Liang, S., Du, Y., Chen, W., & Liu, Z. (2026). Order Allocation and Emergency Transportation Decisions Amid Multi-Source Procurement Disruptions.
17. Zhao, R., Liu, W., Sha, Z., Su, N., Zhang, Y., & Long, Y. (2026). Culturally uneven urban perception in large language models [Preprint]. *arXiv*. <https://doi.org/10.48550/arXiv.2604.20048v3>.
18. Yuan, Y., Huang, J., Yin, J., & Xu, T. (2026). Confidence Calibration and Semantic Error Analysis for Ambiguous Coreference Resolution in User-generated Social Media Text. Available at SSRN 7393238.

19. Zhang, G., Xu, Y., Zhang, M., Wang, S., & Lin, N. (2021). The brain network in support of social semantic accumulation. *Social cognitive and affective neuroscience*, 16(4), 393-405.
20. Bao, Y., & Wang, H. (2026). Large-scale Metrics Migration: A Systematic Approach to Rebuilding Production Measurement Infrastructure. Available at SSRN 7185660.
21. Han, Z., Chen, W., Han, Y., Mao, R., & Qin, J. (2026). Fast Diversified Top-k Rule Discovery via User-Guided Embeddings. *IEEE Transactions on Knowledge and Data Engineering*.
22. Bao, Y., & Wang, H. (2026). Orion: A High-Throughput, Fault-Tolerant Event Routing Architecture for Hyperscale Data Streams. *Fault-Tolerant Event Routing Architecture for Hyperscale Data Streams* (January 01, 2026).
23. Wang, S., Feng, Y., & Fang, X. (2026, May). A Large Language Model-Enabled Multi-Agent Collaboration Method for Complex Task Solving. In *2026 6th International Symposium on Computer Technology and Information Science (ISCTIS)* (pp. 253-256). IEEE.
24. Chaoyong Ma, Nan Si, Xuewei Song, Chen Liang and Haoming Luo, A Bearing Fault Diagnosis Method Based on Concise Empirical Wavelet Transform, *International Journal of Comprehensive Engineering*, 2023, 12(1), 1-6.
25. Yuyan Li, Lin Chen, Mengyu Yang, Qiaomei Li and Taifu Li, Polarization Converters Composed of Metasurfaces with Independent Manipulation of Polarization and Phase, *International Journal of Comprehensive Engineering*, 2023, 12(1), 7-14.
26. Zhao, Z., & Welsch, R. E. (2024). Hierarchical reinforced trader (hrt): A bi-level approach for optimizing stock selection and execution. *arXiv preprint arXiv:2410.14927*.
27. Jiaxun Du, Zhenbao Fu, Jiachen Liu, Huaqing Wang, Zhentao Ge and Liuyang Song, Bearing Fault Diagnosis Method Based on Attention Residual Network Under Small Sample Condition, *International Journal of Comprehensive Engineering*, 2023, 12(1), 15-24.
28. Yang, Z., Alexandrova, A. N., & Sautet, P. (2026). Modeling CO₂ Hydrogenation to Methanol on an Ensemble of Inverse ZrO₂ on Cu Catalytic Sites: Mechanism, Reactivity, and Deactivation. *Angewandte Chemie*, e5448247.
29. Jiawei Lin, Changkun Han, Liuyang Song, Wei Lu and Huaqing Wang, Time Domain Feature Enhancement Model Based on Correlated B-spline Wavelet Base Convolutional Sparse, *International Journal of Comprehensive Engineering*, 2023, 12(1), 25-33.
30. Zhao, Z., & Welsch, R. E. (2026). Point-in-Time Financial RAG with Frozen LLMs and Market-Feedback Adaptive Retrieval. *arXiv preprint arXiv:2605.31201*.
31. Quan, Q., Gao, Y., & Wang, Q. (2025). The impact of teacher emotional support on students' engagement in AI-mediated English learning environments: The mediating role of resilience and self-efficacy. *Acta Psychologica*, 260, 105766.
32. Wang, Z., Yu, J., Liu, H., Zheng, Z., Jin, Y., Li, S., ... & Zhang, K. (2025, July). Generative Music Models' Alignment with Professional and Amateur Users' Expectations. In *Findings of the Association for Computational Linguistics: ACL 2025* (pp. 6909-6920).

33. Xu, Y., Sun, Y., Zhai, B., Li, M., Liang, W., Li, Y., & Du, S. (2025, April). Zero-shot video moment retrieval via off-the-shelf multimodal large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 39, No. 9, pp. 8978-8986).
34. Zheng, X., Chen, X., Gong, S., Griffin, X., & Slabaugh, G. (2025, September). Xfmamba: Cross-fusion mamba for multi-view medical image classification. In *International Conference on Medical Image Computing and Computer-Assisted Intervention* (pp. 672-682). Cham: Springer Nature Switzerland.
35. Su, J., Cai, C., Zhu, F., He, C., Xu, X., Guan, D., & Si, C. (2024, September). Momentum auxiliary network for supervised local learning. In *European Conference on Computer Vision* (pp. 276-292). Cham: Springer Nature Switzerland.
36. Su, J., Zhu, F., Shi, H., Han, T., Qiu, Y., Luo, J., ... & Gao, J. (2025). MAN++: Scaling Momentum Auxiliary Network for Supervised Local Learning in Vision Tasks. *arXiv preprint arXiv:2507.16279*.
37. Li, P., Zhang, H., Li, W., Huang, D., Guo, Z., Chen, J., ... & Koshizuka, N. (2025). GeoAvatar: A big mobile phone positioning data-driven method for individualized pseudo personal mobility data generation. *Computers, Environment and Urban Systems*, 119, 102252.
38. Lyu, Y., Liu, K., Cao, Z., Luo, Y., Yin, L., Yang, T., & Chen, Z. (2026). FLAIR: An LLM-integrated multi-layer framework for fine-grained activity type inference in human trajectory data. *Transportation Research Part C: Emerging Technologies*, 190, 105787.
39. X. Chen, C. Huang, G. Chen, D. Feng and P. Xiao, "The Communication and Computation Trade-Off in Wireless Semantic Communications," in *IEEE Wireless Communications Letters*, vol. 14, no. 7, pp. 2259-2263, July 2025, doi: 10.1109/LWC.2025.3569205.
40. Cui, J., Zhang, G., Chen, Z., & Yu, N. (2022). Multi-homed abnormal behavior detection algorithm based on fuzzy particle swarm cluster in user and entity behavior analytics. *Scientific Reports*, 12(1), 22349.
41. Wang, T., & Xia, Z. (2025). Stability of In-Context Learning: A Spectral Coverage Perspective. *arXiv preprint arXiv:2509.20677*.