A Human–Machine Collaborative Tuning Framework for Triton Kernel Optimization on SIMD Platforms

Xulin Zhou[1,2], Hongbin Zhang[1], Mingjie Xing[1]
[1] Institute of Software, Chinese Academy of Sciences, Beijing, China
[2] University of Chinese Academy of Sciences, Beijing, China
{zhouxulin2023, hongbin2019, mingjie}@iscas.ac.cn

***Abstract*—Single Instruction, Multiple Data (SIMD) technology enhances performance through parallel data processing on CPUs. SIMD platforms are widely adopted across domains ranging from high-performance computing to AI inference. As modern AI workloads increasingly rely on Python-based kernel frameworks to maintain usability and benefit from automatic tuning, Triton has emerged as a representative solution. However, Triton's auto-tuning mechanism, designed primarily for NVIDIA GPUs, fails to effectively exploit the architectural features of SIMD CPUs, creating a significant performance gap on these platforms.**

To address this problem, we introduce a human–machine collaborative design tailored for Triton kernel tuning on SIMD platforms. This design improves both development efficiency and performance by capturing high-level SIMD optimization intent from human users and integrating it seamlessly into machine framework tuning. Based on this collaborative design, we develop a tuning framework composed of a front-end for user intent recognition and a back-end for user-guided, SIMD-aware tuning. Experiments on x86 and RISC-V platforms show an average performance improvement of 31.7% over native Triton tuning, with tuning cost reduced by up to 75.0%.

***Index Terms*—SIMD, Kernel Optimization, Auto-Tuning, Triton, Human-Machine Collaborative Design.**

I. Introduction

SIMD is critical for enhancing performance through parallel data processing on CPUs. SIMD platforms are widely employed in high-performance computing domains such as image processing and scientific computing, and are increasingly leveraged for deep learning model inference[1][2].

Unlike traditional C++ kernel optimization approaches [3][4], the modern AI ecosystem increasingly relies on Python-based kernel frameworks to maintain tight integration with PyTorch [5] and to significantly lower the barrier for kernel development. This shift reflects the growing need for a higher-level and more accessible programming interface while preserving compatibility with existing AI toolchains. Among these frameworks, Triton[6] has emerged as a representative, allowing developers to write kernels in a Python-like language while automatically tuning parameters to boost performance. Compared to manual tuning illustrated in Figure 1a, such automatic tuning avoids high development complexity[7][8].

However, Triton's kernel performance remains insufficient on SIMD platforms. As shown in Figure 1b, its tuning mechanisms primarily depend on framework capabilities, leaving limited room for effective human involvement. While Triton provides comprehensive capabilities for NVIDIA GPUs, it lacks sophisticated tuning support for SIMD CPUs, resulting in suboptimal performance.

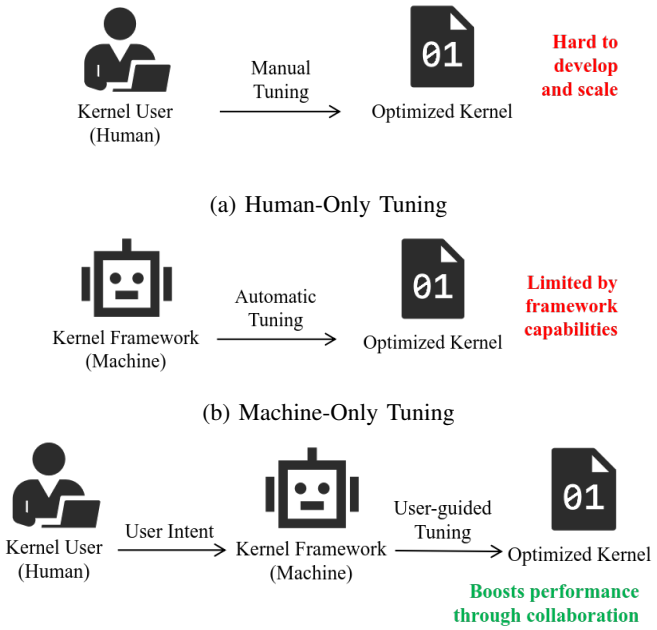


Fig. 1: Different design paradigms of kernel tuning. (a) Humans have full control over tuning. (b) The machine has full control over tuning. (c) Humans provide high-level intent to help the machine perform collaborative tuning.

To address this limitation, we introduce a novel human–machine collaborative design for Triton kernel tuning on SIMD platforms. As illustrated in Figure 1c, this design balances ease of development with high performance. Our framework consists of two coordinated components: a front-end for user intent recognition and a back-end for user-guided tuning. The front-end collects user hints and defines the tuning task, while the back-end performs hint-driven analysis and applies tuning algorithms to determine optimal configurations.

```python
@triton.jit
def add_kernel(x_ptr, y_ptr, output_ptr, n_elements,
BLOCK_SIZE):
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y
    tl.store(output_ptr + offsets, output, mask=mask)

```

Fig. 2: Example Triton kernel implementation.

We evaluate our framework against human-only and machine-only tuning on both x86 and RISC-V SIMD platforms. Experimental results demonstrate an average performance gain of 31.7% over the native Triton tuning approach on the x86 AVX512 architecture, with a comparable gain observed on the RISC-V Vector Extension. Notably, our approach reduces tuning costs by up to 75.0% compared to native Triton tuning[9] and by over 96.3% compared to TVM Ansor tuning[10].

In summary, our main contributions are as follows:

- A front-end that interprets user intent and translates it into actionable guidance for back-end tuning.
- A back-end that incorporates user hints, analyzes their impact on the search space, and determines optimal configurations through a hint-guided tuning algorithm, with comprehensive integration and validation in Triton on SIMD platforms.
- A novel human–machine collaborative framework integrating the proposed front-end and back-end for Triton kernel tuning on SIMD platforms.

II. Background and Related Work

A. Triton Kernel Optimization and Its Scope

Triton is a user-friendly, Python-based kernel framework designed to bridge the gap between high-level productivity and low-level performance. An example Triton kernel is shown in Figure 2. Its growing adoption is driven by two factors. First, its integration with the PyTorch ecosystem ensures strong back-end support and accessibility. Second, while originally developed for NVIDIA GPUs, Triton has been extended to a range of platforms, including AMD GPUs[11] and Ascend NPUs[12]. Triton also has experimental support on x86[9], ARM[13], and RISC-V platforms[14]. Despite this broad portability, Triton's performance on CPU SIMD platforms remains suboptimal. This paper focuses on enhancing Triton kernel optimization specifically for SIMD platforms, building upon the experimental CPU support introduced in [9].

B. Tuning Paradigms on SIMD Platforms

Optimizing AI kernels for SIMD platforms is critical, given their prevalence in edge devices, embedded systems, and other resource-constrained environments. Existing approaches can be categorized into three paradigms:

Human-only tuning. Representative efforts include `llama.cpp` [15], which relies on hand-tuned assembly and intrinsics, and established libraries such as OpenBLAS [16], OpenCV [17], and Intel oneDNN [18], all of which incorporate extensive platform-specific manual optimizations.

Machine-only tuning. Frameworks such as AutoTVM [19][20] explore parameterized schedules using feature-driven cost models, while Ansor [10] expands the search space through sampling guided by learned performance predictors. Tilelang [21] and Roller[22] provide high-level abstractions that enable experts to expose fine-grained scheduling.

Human–machine collaborative tuning. Examples include OpenMP [23], which allows programmers to provide high-level hints, and Triton's decorators such as `@heuristic` and `@tune` [11]. However, these interfaces are not designed for collaborative tuning on SIMD platforms. This work fills that gap by introducing a framework that integrates user guidance into automated Triton kernel tuning for SIMD platforms. In the following method section, we show how this guidance is formalized into explicit parameter constraints and then validated experimentally.

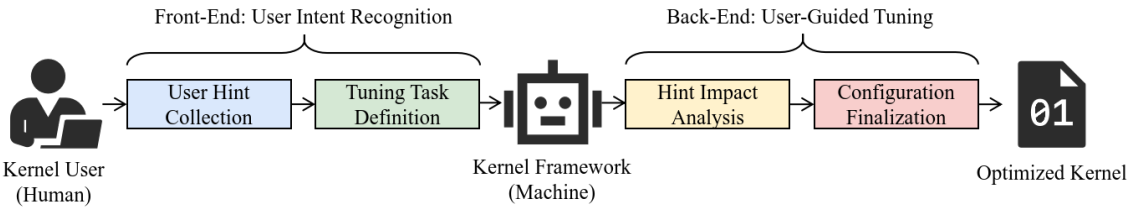
III. System Overview

Our human–machine collaborative kernel tuning framework integrates high-level human intent with automated framework-driven optimization. As shown in Figure 3a, the system is composed of two components: a front-end for user intent recognition and a back-end for user-guided tuning. Together, these components enable the system to incorporate SIMD-related expertise without requiring users to engage in low-level performance engineering, while still leveraging automated search for high-quality optimization results.

Figure 3b demonstrates the kernel tuning pipeline for an example kernel. The user-guided tuning process begins with user intent recognition, where optimization hints are captured via a custom decorator "*@heuristic_intent*" and integrated with the kernel implementation. These hints then inform the tuning task definition, which establishes three key parameters for optimization. During hint impact analysis, user hints dynamically reshape the search space of these parameters. Finally, configuration finalization leverages this refined search space to automatically discover superior parameter configurations that outperform machine-only tuning, demonstrating collaboration between human insight and automated optimization.

IV. Front-End: User Intent Recognition

The front-end bridges user intent to the tuning framework through two modules: user hint collection and tuning task definition. The former interprets and incorporates user intent into the tuning process, while the latter formalizes the tuning task by extracting key parameters for the back-end.



(a) Collaborative Tuning Framework Overview

User Intent Recognition	Tuning Task Definition	Hint Impact Analysis	Configuration Finalization
Input Triton kernel: # Pass user hint to Triton custom decorator @heuristic_intent(input_size = 128, thread_count_max = none, hardware = 'avx512') # Implement kernel with Triton language @triton.jit def kernel(...): # implement kernel with Triton...	**Tuning parameters:** thread_count vector_size block_size **User hints for tuning:** Workload-specific: input_size, thread_count_max; Hardware-specific: reg_width	**Before user-guided:** thread_count ∈ {8} vec_size ∈ {1, 2, 4, 8, 16} block_size ∈ {1, 2, 4, 8, 16} **After user-guided:** thread_count ∈ {2, 4, 8} vec_size ∈ {8, 16, 32, 64} block_size ∈ {8, 16, 32, 64}	**Before user-guided:** thread_count = 8 vec_size = 16 block_size = 16 **After user-guided:** thread_count = 4 vec_size = 32 block_size = 64

(b) An Example of Tuning Framework Pipeline

Fig. 3: Human-Machine Collaborative Tuning Framework for Triton Kernel Optimization

A. User Hint Collection

We collect two types of user hints: workload-specific and hardware-specific. Workload hints include kernel workload size and the maximum number of threads, whereas hardware hints cover features such as vector register width and register count. Users are not required to specify exact numerical values. As shown in Figure 3b, if the thread count is omitted, it defaults to the maximum available. For hardware hints, users only need to input the SIMD platform (e.g., `hardware='avx512'`); our module automatically retrieves the architectural parameters, which are fixed for common SIMD platforms. These hints are represented as structured variables (e.g., $input\_size$, $thread\_count\_max$, and register properties), so that they can be directly consumed by the equations and algorithms in the back-end.

B. Tuning Task Definition

We define the tuning task using three kinds of parameters: vec_size for vector configuration, $thread\_count$ for workload division, and $block\_size$ for tiling granularity. Figure 4 illustrates these parameters in practice. For vector addition in Figure 4a, the 1D input is divided into segments of size $block\_size$, with each segment assigned to an independent thread. Within each thread, data is processed using vector instructions of width vec_size, thus requiring vector configuration. In other kernels, parameters at the same parallelism level may involve multiple variables. Figure 4b demonstrates matrix multiplication across dimensions M, N, and K. The output is partitioned into 2D blocks (of size $block\_size\_M$ and $block\_size\_N$) distributed among threads, while the K dimension (with size $block\_size\_K$) is reduced. Higher-dimensional cases can be represented similarly. This unified parameterization enables the back-end to reason about user intent in a kernel-agnostic yet formally analyzable way.

V. Back-End: User-Guided Tuning

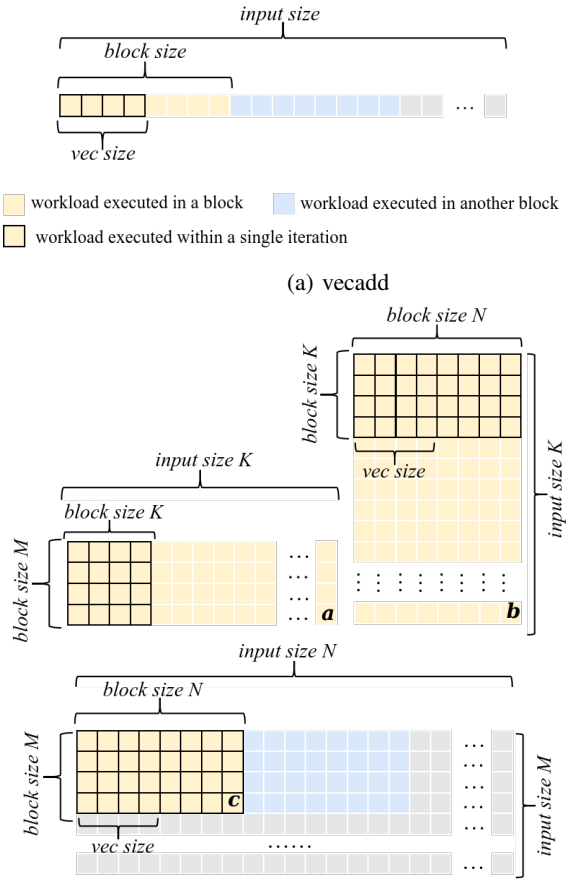
The back-end implements the user-guided tuning through a two-stage methodology: hint impact analysis and configuration finalization. The outcome is an optimized kernel that reflects a collaboration work of human intent and machine tuning.

A. Hint Impact Analysis

We analyze how user hints including workload-specific hints and hardware-specific hints influence the tuning parameters we previously defined in tuning task definition of the front-end. We first consider how user hints affect vec_size. The most relevant user hint for vec_size is vector resources. Let reg_width denote the bit width of a vector register, and ele_width the bit width of the data element type, the baseline vector width supported by a single vector register is:

$$vec\_size\_basic = \frac{reg\_width}{ele\_width} \tag{1}$$

In practice, vector operations can span multiple registers. If several registers are grouped during a single instruction (e.g., via loop unrolling at the IR level), then vec_size becomes a multiple of vec_size_basic. Hence, vec_size_basic represents the minimum possible value for vec_size.



(a) vecadd

(b) matmul

Fig. 4: Illustration of Tuning Parameters for Different Kernels

Next, we examine how user hints affect $thread\_count$. Let $block\_count$ denote the total number of blocks divided from the input workload. In 1D blocking scenario shown in Figure 4a, since the blocking technique divides the input workload into blocks of size $block\_size$, the total number of blocks is $\frac{input\_size}{block\_size}$. In 2D blocking scenario shown in Figure 4b, we have:

$$block\_count = \frac{input\_size\_N}{block\_size\_N} \times \frac{input\_size\_M}{block\_size\_M} \quad (2)$$

$$thread\_count \leq min(block\_count, thread\_count\_max) \quad (3)$$

Noting that $thread\_count$ is closely related to $block\_size$, and the user hints influencing it primarily include workload-specific hints such as $input\_size$ and $thread\_count\_max$.

B. Configuration Finalization

Building on the previous analysis, we outline the tuning algorithm for determining the configuration of tuning parameters. We use the 2D blocking case in Figure 4b as a running example; the 1D case in Figure 4a is treated as a special instance where non-vector block dimensions are fixed to 1.

Algorithm 1 Tuning algorithm for finalizing configuration

Input: User hints
Output: Final configuration of tuning parameters
Step 1: Initialization
Compute vec_size_basic
$S \leftarrow \emptyset$
Step 2: Calculate block size range
$B_n \leftarrow \{b_n \in \mathbb{N}^+ \mid b_n = p \cdot vec\_size\_basic,\ p \in \mathbb{N}^+,\ b_n \leq input\_size\_N\}$
$B_m \leftarrow \{b_m = 2^q \mid q \in \mathbb{N}, 1 \leq b_m \leq input\_size\_M\}$
$B_k \leftarrow \{b_k = 2^r \mid r \in \mathbb{N}, 1 \leq b_k \leq input\_size\_K\}$
$\mathbf{B} \leftarrow B_m \times B_n \times B_k$
Step 3: Calculate vec size and thread count ranges
for each $block\_size \in \mathbf{B}$ **do**
    Compute $block\_count$
    $V \leftarrow \{v \in \mathbb{N}^+ \mid v = k \cdot vec\_size\_basic,\ k \in \mathbb{N}^+,\ v \leq block\_size\_N\}$
    $thread\_count\_up \leftarrow \min(thread\_count\_max, block\_count)$
    $T \leftarrow \{t \in \mathbb{N}^+ \mid 1 \leq t \leq thread\_count\_up\}$
    $S \leftarrow S \cup \{\langle block\_size, V, T\rangle\}$
end for
Step 4: Finalize tuning parameter configuration
$s_{best} \leftarrow \underset{s \in S}{\text{argmax}}\ \text{Performance}(s)$
return s_{best}

We first determine the tuning order so that earlier decisions can constrain later ones. We argue that $block\_size$ should be tuned first, followed by vec_size and $thread\_count$. This is because blocking is the key connector between SIMD execution and multi-threading: once the block size is fixed, the choices of vec_size and $thread\_count$ jointly determine vector utilization and workload partitioning.

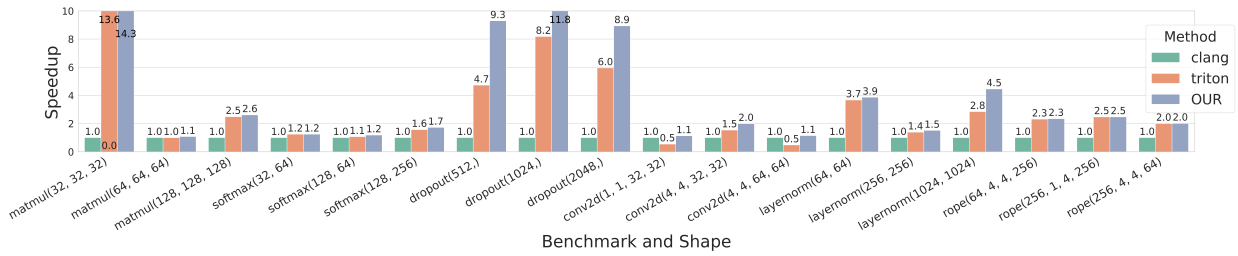
Next, we define the search ranges. The vectorized dimension of $block\_size$ (e.g., $block\_size\_N$) is restricted to vec_size_basic or its multiples to ensure SIMD alignment, while other block dimensions grow in powers of two within the limits of $input\_size$ to improve memory locality. With the $block\_size$ range established, the feasible range of vec_size is determined—bounded below by vec_size_basic and above by the vectorized block dimension. The range of $thread\_count$ then follows from Equation (3). Algorithm 1 summarizes the full procedure.

VI. Experimental Results

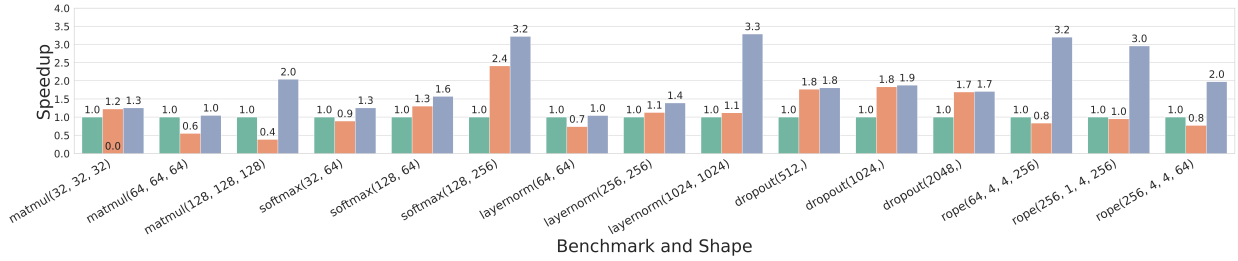
Our experiments consist of three parts. First, we compare the performance of our collaborative tuning approach with native Triton tuning. Second, we evaluate it against other machine-only methods in terms of both performance and tuning overhead, as well as against human-only methods in terms of performance. Finally, we analyze how thread count shapes the selection of optimal tuning parameters and show that our collaborative tuning uncovers new, thread-aware search spaces that native Triton tuning fails to explore.

A. Experimental Setup

We evaluate six representative AI kernels—*matmul*, *softmax*, *conv2d*, *layernorm*, *dropout*, and *rope*—covering a broad range of computation patterns. Specifically, *matmul* and *conv2d* are computation-intensive kernels, *layernorm* is a memory-bound kernel, *dropout* involves stochastic masking,

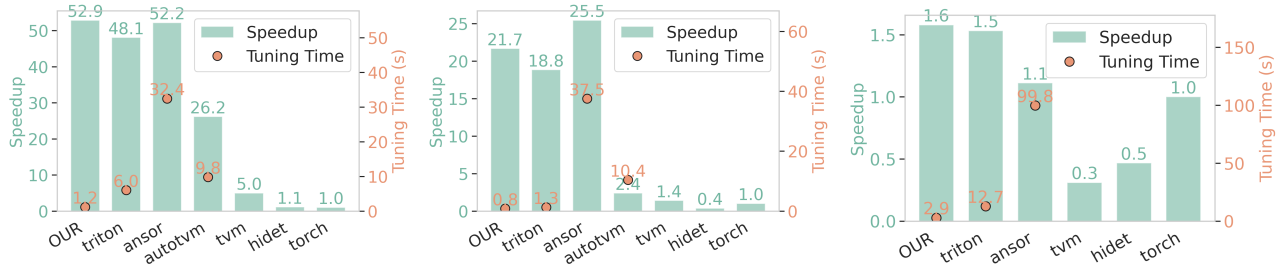


(a) Performance comparison on x86 platform.



(b) Performance comparison on RISC-V platform.

Fig. 5: Performance speedup of our approach compared with native Triton tuning, normalized to Clang's execution time.



(a) matmul (b) softmax (c) conv2d

Fig. 6: Performance speedup and tuning time cost with different tuning approaches. The speedup is normalized to Torch's execution time. Since TVM, Hidet, and Torch use manual implementations or operator libraries, they do not require tuning.

softmax represents a fused kernel, and *rope* is a positional encoding technique widely used in Transformer models. These carefully selected kernels cover a range of representative workloads in AI acceleration.

We evaluate our approach and other tuning methods on two multi-core processors, whose specifications are listed in Table I. Each kernel undergoes 25 warm-up runs, followed by 100 executions to compute the average runtime.

TABLE I: Hardware Information

Machine	Platform	SIMD Extension	Available Threads
Xeon(R) Gold	x86	AVX-512	12
SpacemiT K1	RISC-V	Vector	8

B. Comparison with native Triton tuning

We compare our auto-tuning method against Clang's auto-vectorization and native Triton grid search. The x-axis denotes the evaluated kernels and shapes, and the y-axis reports the speedup over Clang. Experiments are conducted on x86 and RISC-V SIMD platforms.

As shown in Figure 5, our method delivers substantial gains on x86, achieving an average improvement of 31.7% and up to 97.8% for dropout. RISC-V results also show superior performance. However, for the x86 rope and RISC-V dropout benchmarks, our performance matches Triton's, likely because Triton's default settings are already near-optimal and insensitive to input variations. Overall, our method consistently outperforms native Triton tuning across both architectures.

C. Comparison with Other Tuning Approaches

We further compare our tuning framework with machine-only methods (Triton, Ansor, AutoTVM) and Python-based human-only methods (TVM, Hidet). The y-axis reports speedup over PyTorch.

As shown in Figure 6, our method surpasses Triton across all machine-only baselines, achieving up to 15.4% higher performance on softmax and rivaling Ansor on matmul and conv2d. Moreover, it reduces tuning cost by up to 96.3%

TABLE II: Comparison of tuning parameters with different thread-setting methods.

Benchmark	Shape	Tuning Parameters					Native Speedup	Our Speedup	Perf. Gain
		T1 best	T4 best	T8 best	Native	Ours			
matmul	(64,64,64)	(32,32,32)	(8,4,8)	(16,4,4)	(32,32,32)	(4,32,4)	1.42	1.43	1.13%
matmul	(128,128,128)	(32,16,4)	(16,32,8)	(16,4,32)	(32,16,4)	(32,8,32)	2.43	3.10	27.19%
softmax	(128,64)	(64,128)	(1024,64)	(64,128)	(64,128)	(64,8)	1.30	1.80	37.80%
softmax	(128,256)	(64,8)	(64,8)	(4096,16)	(64,8)	(64,8)	1.00	1.90	90.86%
conv2d	(4,4,32,32)	(2,32)	(2,32)	(4,4)	(2,32)	(1,16)	1.30	1.42	8.58%
conv2d	(4,4,64,64)	(4,8)	(1,4)	(1,16)	(4,8)	(4,8)	2.19	2.19	-
layernorm	(256,256)	(256,32)	(256,64)	(256,8)	(256,32)	(256,4)	1.31	1.57	19.93%
layernorm	(1024,1024)	(1024,16)	(1024,16)	(1024,4)	(1024,16)	(1024,8)	2.48	4.42	77.87%
dropout	1024	(1024,16)	(4096,16)	(1024,64)	(1024,16)	(1024,64)	11.97	14.97	25.08%
dropout	2048	(4096,64)	(4096,64)	(4096,32)	(4096,64)	(4096,64)	10.22	10.22	-
rope	(256,1,4,256)	(256,32)	(256,64)	(256,256)	(256,32)	(256,8)	1.10	2.87	159.82%
rope	(256,4,4,64)	(256,64)	(256,32)	(256,32)	(256,64)	(256,32)	1.31	3.07	133.15%

compared with Ansor and up to 75.0% compared with Triton, except for softmax where the reduction is 38.4% due to Triton's smaller search space and initialization overhead. Thus, our approach achieves competitive performance with substantially lower tuning cost.

Against human-only methods, our approach consistently outperforms manually tuned implementations, particularly on matmul and softmax where it yields order-of-magnitude improvements. This is likely because manual tuning strategies are often GPU-oriented and do not align well with CPU constraints such as limited cache and vector registers.

D. Interactions Between Thread Count and Tuning Parameters

Our results show that optimal tuning parameters vary significantly with thread count. As illustrated in Table II, different thread configurations (`T1`, `T4`, `T8`) lead to distinct parameter choices that maximize performance. All reported values are normalized to Clang, and "Perf. Gain" measures improvements over Triton's defaults. As thread count increases, the per-thread workload shrinks, shifting the preferred blocking, vectorization, and other tuning decisions. Because Triton reuses parameters tuned for single-thread execution, it fails to capture these shifts, whereas our method re-selects parameters for each thread setting and achieves up to 159.82% improvement.

VII. Conclusion and Future Work

To overcome Triton's limited SIMD support and suboptimal performance on SIMD platforms, we propose a human–machine collaborative framework for kernel tuning. By integrating a front-end that captures SIMD optimization intent with a back-end that leverages this intent for hint-guided tuning, our approach narrows the performance gap, enables thread-aware exploration beyond native Triton tuning's capabilities, and simultaneously reduces tuning cost.

Looking ahead, we will extend our evaluation to more sophisticated attention kernels and end-to-end deep learning models, and we will continue improving both user-intent recognition and tuning algorithms. Moreover, these strategies can further benefit from incorporating LLM-based knowledge suggestions to strengthen collaborative kernel optimization.

References

[1] Evangelos Georganas et al. "Anatomy of high-performance deep learning convolutions on SIMD architectures". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 830–841.

[2] Yifei He and Stefano Markidis. "High-performance fft code generation via mlir linalg dialect and simd micro-kernels". In: *2024 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2024, pp. 155–165.

[3] C++ Standard Committee. *std::experimental::simd*. https://en.cppreference.com/w/cpp/experimental/simd. Accessed: 2024-05-15. 2023.

[4] Vijay Kandiah et al. "Parsimony: Enabling SIMD/Vector Programming in Standard Compiler Flows". In: *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*. 2023, pp. 186–198.

[5] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019.

[6] Philippe Tillet, H. T. Kung, and David Cox. "Triton: an intermediate language and compiler for tiled neural network computations". en. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. Phoenix AZ USA: ACM, June 2019, pp. 10–19.

[7] Yishen Chen et al. "VeGen: a vectorizer generator for SIMD and beyond". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 902–914.

[8] Samuel Thomas and James Bornholt. "Automatic Generation of Vectorizing Compilers for Customizable Digital Signal Processors". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 2024, pp. 19–34.

[9] OpenAI. *Triton-CPU*. 2023. URL: https://github.com/triton-lang/triton-cpu.

[10] Lianmin Zheng et al. "Ansor: Generating High-Performance Tensor Programs for Deep Learning". en. In: 2020, pp. 863–879.

[11] OpenAI. *Triton*. 2019. URL: https://github.com/triton-lang/triton.

[12] Huawei Technologies Co., Ltd. *Triton-Ascend*. 2024. URL: https://ascend.github.io/triton-ascend/.

[13] Ltd Huawei Technologies Co. *triton-openeuler*. 2024. URL: https://github.com/openeuler-mirror/triton-cpu.

[14] Terapines Technology(Wuhan) Co,.Ltd. *AI-Benchmark*. 2024. URL: https://github.com/Terapines/AI-Benchmark.

[15] Georgi Gerganov. *llama.cpp*. 2023. URL: https://github.com/ggml-org/llama.cpp.

[16] Xianyi Zhang, Xi Wang, and Yunquan Zhang. "OpenBLAS: A High Performance BLAS Library on Loongson 3A CPU". In: *Journal of Software* 22.zk2 (2012), pp. 208–216.

[17] Gary Bradski. "The openCV library." In: *Dr. Dobb's Journal: Software Tools for the Professional Programmer* 25.11 (2000), pp. 120–123.

[18] Jianhui Li et al. "onednn graph compiler: A hybrid approach for high-performance deep learning compilation". In: *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2024, pp. 460–470.

[19] Tianqi Chen et al. "{TVM}: An automated {End-to-End} optimizing compiler for deep learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 578–594.

[20] Tianqi Chen et al. "Learning to optimize tensor programs". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada: Curran Associates Inc., 2018, pp. 3393–3404.

[21] Lei Wang et al. "TileLang: A Composable Tiled Programming Model for AI Systems". In: *arXiv preprint arXiv:2504.17577* (2025).

[22] Hongyu Zhu et al. "ROLLER: Fast and Efficient Tensor Compilation for Deep Learning". en. In: 2022, pp. 233–248.

[23] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.