LP-GEMM: Integrating Layout Propagation into GEMM Operations

César G. Carneiro[1], **Lucas F. Alvarenga**[1], **Guido C. Araujo**[1], **Sandro Rigo**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Campinas – SP – Brazil

c261031@dac.unicamp.br, lucas.silva@ic.unicamp.br,

{guido, srigo}@unicamp.br

Abstract. *High-Performance Computing and Machine Learning workloads spend much of their time in chains of dependent Matrix Multiplications (GEMMs), but GEMM implementations, such as those in most BLAS libraries, optimize each call in isolation, missing cross-call redundancy. We present LP-GEMM, a decomposition of the OpenBLAS GEMM kernel into Initial, Intermediate and Ending kernels propagating the packed layout through a chain, with BLAS-correct endpoints. LP-GEMM on 14 GEMM chains using one thread of an x86 (AVX-512) CPU, and excluding conversions at the chain's end, is within 7% of the fastest of five libraries on 8 of the 14 chains and faster on three. In a parallelism experiment, LP-GEMM scales like OpenBLAS across 1–24 threads.*

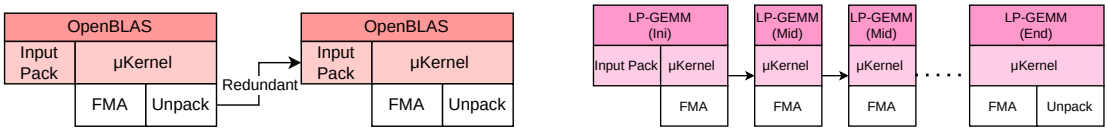
1. Introduction

Scientific Computing and modern Machine Learning (ML) workloads [Vaswani et al. 2017, Meta's Llama 3 team 2024, Devlin et al. 2019] are dominated by large volumes of matrix data, computed either as a single Matrix Multiplication (MM), a sequence of dependent MM operations, or a batch of independent MM operations. Operations not immediately related to them, such as convolutions, can also be reduced to MM once the data is properly reorganized (*e.g.*, img2col) before execution.

Matrix Multiplication is generally implemented as the General Matrix Multiply (GEMM) operation, which generalizes it with a scaling parameter α and a scaling-accumulation parameter β. Given its relevance, GEMM has been optimized at the hardware level [Genc et al. 2021, Li et al. 2024b, Kim et al. 2024], the code level [Goto and van de Geijn 2008, Zhang et al. 2025], and the algorithmic level [Alman et al. 2025]. Hardware vendors also ship closed-source, highly tuned BLAS-compliant libraries, such as Intel MKL [Intel Corporation 2024] and NVIDIA cuBLAS [NVIDIA Corporation 2024].

OpenBLAS [1] is the best-known open-source effort in this space, inheriting the approach of Goto and van de Geijn [Goto and van de Geijn 2008]: a sequence of tiling, packing, and loop-unrolling steps applied to the naive MM kernel to exploit the memory hierarchy. However, the BLAS API makes no assumptions about subsequent operations, so a Goto-style implementation packs its inputs and restores its output to a canonical memory layout independently on every call. This is wasteful whenever GEMMs are

[1]https://github.com/OpenMathLib/OpenBLAS



(a) Consecutive GEMM executions using OpenBLAS.

(b) Consecutive GEMM executions using LP-GEMM.

Figure 1. Comparison of OpenBLAS and LP-GEMM kernel execution of consecutive GEMM operations (two GEMMs shown; the same pattern generalizes to an arbitrary number). The relative block sizes do not represent execution time; weight packing is omitted for clarity.

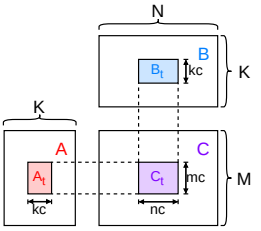
chained, as they routinely are in the MLP and Transformer blocks of modern ML models, since every MM redundantly repeats the library's full pack/compute/unpack cycle even though the previous GEMM's output already exists in a compatible packed form. This is especially relevant for OpenBLAS.

This work observes that when MM operations are executed sequentially, redundant packing steps are performed for each of them, as shown in Fig. 1a. Therefore, we propose the Layout Propagation GEMM (LP-GEMM), a decomposition of the OpenBLAS GEMM kernel into three specialized kernels that avoid executing redundant packing steps. Specifically, the GEMM operations are divided into: (1) the Initial GEMM (ini-GEMM), which performs packing and propagates its output layout; (2) the intermediate GEMM (mid-GEMM), which receives the data already packed from ini-GEMM and retains this layout for the next operations; and (3) the Ending GEMM (end-GEMM), which unpacks the data into the original layout. Fig. 1b shows how LP-GEMM can simplify the BLAS GEMM kernel, removing unnecessary operations by leveraging layout propagation between operations.

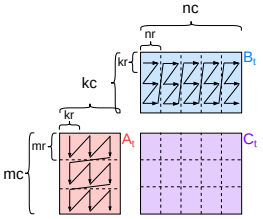
We evaluate LP-GEMM on an AVX-512 x86 platform across 14 consecutive-GEMM chains from HPC, MLP, and Transformer workloads, at 1 to 24 threads, against OpenBLAS and four other libraries. Single-threaded, and excluding the layout conversions at the ends of the chain, LP-GEMM is within 7% of the fastest library on 8 of the 14 workloads and faster on three, while using no more instructions per FLOP than OpenBLAS; its clear losses are on $N{=}1$ shapes. With more threads it scales like OpenBLAS, and its cost relative to OpenBLAS does not depend on thread count.

The main contributions of this work are as follows: (1) the LP-GEMM decomposition of the OpenBLAS GEMM kernel into ini-GEMM, mid-GEMM, and end-GEMM, which propagates the packing layout across a chain with BLAS-correct endpoints. (2) an evaluation against five libraries on 14 chains, with absolute throughput and hardware counters, showing where propagation wins, ties, and loses, and why. (3) a thread-scaling study (1–24 threads) showing that LP-GEMM's cost relative to OpenBLAS stays in a flat band, so propagation does not harm parallel scalability.

Section 2 introduces the background, Section 3 describes LP-GEMM, Section 4 presents the results, Section 5 reviews the literature, and Section 6 concludes.



(a) Outer tiling. Kernel



(b) Inner tiling. micro-kernel (µKernel)

1: **for** $j = 0$ **to** N **step** n_c **do**
2: **for** $l = 0$ **to** K **step** k_c **do**
3: Pack m_c rows of A_T into **sa**
4: **for** $jj = j$ **to** $j + n_c$ **step** n_r **do**
5: Pack n_r columns of B_T into **sb**
6: $C_T \leftarrow A_T \cdot B_T$ {µKernel w/ **sa**, **sb**}
7: **end for**
8: **for** $i = m_c$ **to** M **step** m_c **do**
9: Pack next A_T into buffer **sa**
10: $C_T \leftarrow A_T \cdot B_T$ {µKernel w/ **sa**, **sb**}
11: **end for**
12: **end for**
13: **end for**

(c) GotoBLAS's algorithm.

Figure 2. The GotoBLAS approach. (a) shows how matrices are tiled for the memory hierarchy. (b) depicts how each block A_t and B_t is further tiled and packed for better usage of the processor's registers during C_T computing of µKernel. Finally, (c) shows the GotoBLAS GEMM algorithm.

2. Background

GEneral Matrix Multiplication (GEMM), as described in Equation 1, refers to a standard matrix multiplication of two operands $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$ that produces a matrix $C \in \mathbb{R}^{m \times n}$, extended with two additional scaling parameters α and β.

$$C = \alpha A \times B + \beta C \tag{1}$$

Naive triple-loop implementations underuse the memory hierarchy, reloading operands with little data reuse and, for a row-major B, poor reuse and frequent cache misses. Goto and Van de Geijn [Goto and van de Geijn 2008] addressed this with GotoBLAS's *outer-product* formulation: GEMM is expressed as a sequence of rank-1 updates $C \leftarrow C + A_{:,l} \times B_{l,:}$ that accumulate into register-resident tiles of C using Fused Multiply-Add (FMA) instructions. This relies on two techniques that underlie modern BLAS implementations: **tiling** (or blocking), which adjusts the iteration space to operate on memory-hierarchy-sized blocks rather than the whole dataset [Lam et al. 1991, Wolfe et al. 1995]; and **packing**, which buffers tiled blocks into contiguous memory reordered to match the execution stride, reducing TLB and cache conflict misses [Rohwedder et al. 2023]. For simplicity, we assume $\alpha = 1$ and $\beta = 0$ for the remainder of the paper.

Fig. 2 shows the GotoBLAS GEMM optimization strategy. The outer tiling, referred to as the **kernel** (Fig. 2a), tiles all MM-related dimensions using parameters n_c, m_c, k_c (memory-hierarchy-derived) and n_r, m_r, k_r (register-derived). A **micro-kernel** (µKernel, Fig. 2b) then computes a given C_T tile from blocks A_T and B_T, which are first packed into contiguous buffers before computation. The C_T results are generated in this packed layout, requiring an unpacking step before being written back to memory.

Basic Linear Algebra Subprograms (BLAS) defines a standard set of routines for vector and matrix operations, categorized into three levels: Level 1 (scalar, vector, and vector-vector operations), Level 2 (matrix-vector operations), and Level 3 (matrix-matrix operations); its efficiency, portability, and availability make it the foundation of

most high-quality linear algebra software [The Netlib 2025]. Processor vendors provide hardware-tuned implementations, such as MKL (closed-source) [Intel Corporation 2024] and oneDNN [Li et al. 2024a] for Intel processors and cuBLAS for NVIDIA devices [NVIDIA Corporation 2024]. OpenBLAS is an open-source alternative that extends the GotoBLAS approach to all BLAS API operations across many architectures, with the innermost tiling (*i.e.*, the µKernel) implemented by experts for each target.

This work focuses on the GEMM implementation in OpenBLAS targeting Intel x86 processors, leveraging the AVX-512 vector extension. An OpenBLAS GEMM consists of input packing, computation via the µKernel, and output unpacking to restore the standard memory layout, repeating this full cycle at every GEMM boundary when calls are chained (Fig. 1a). Section 3 details how LP-GEMM's Initial, Intermediate, and Ending kernels (Fig. 1b) avoid this redundancy by propagating the packing layout across the chain instead.

3. Propagating Layout in GEMM Operations

Typical BLAS libraries operate as isolated kernels that always produce outputs in the same layout as the naive implementation, introducing redundant work across GEMM boundaries. LP-GEMM removes that redundancy by letting the packing layouts constructed in earlier kernels be propagated to subsequent consumers of the same matrix.

Fig. 3 shows two layouts OpenBLAS can use for sequential MM, generating $E = D \cdot C$ with $C = A \cdot B$. In the left approach, the output of one MM is the multiplicand of the next; the right approach instead uses matrix transposes so the output becomes the *multiplier* of the next MM, giving a clear common layout between what the first GEMM produces and what the second consumes, as shown by the highlighted tiles in both instances of C_T. A shared layout alone does not guarantee seamless fusion, however: non-GEMM operators interleaved between calls (*e.g.*, activations) may consume a different slice or stride than the one LP-GEMM produces, requiring the operator itself to be adapted.

OpenBLAS's output has a tile structure similar to the multiplier's, as shown by the C_T compute-order arrow in Fig. 4a, enabling the straightforward layout propagation LP-GEMM leverages. The traditional (non-transposed) approach would instead require complex output repacking, adding overhead; this paper therefore assumes transposed matrices throughout.

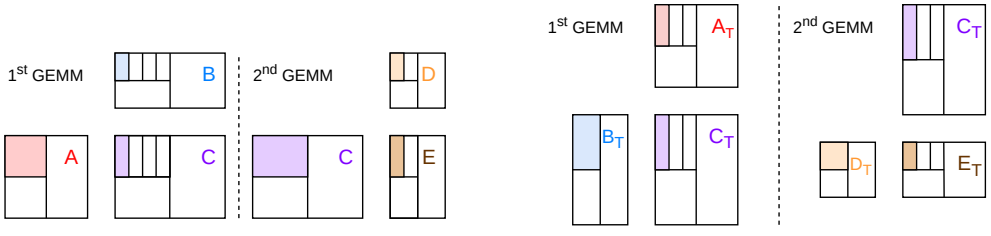
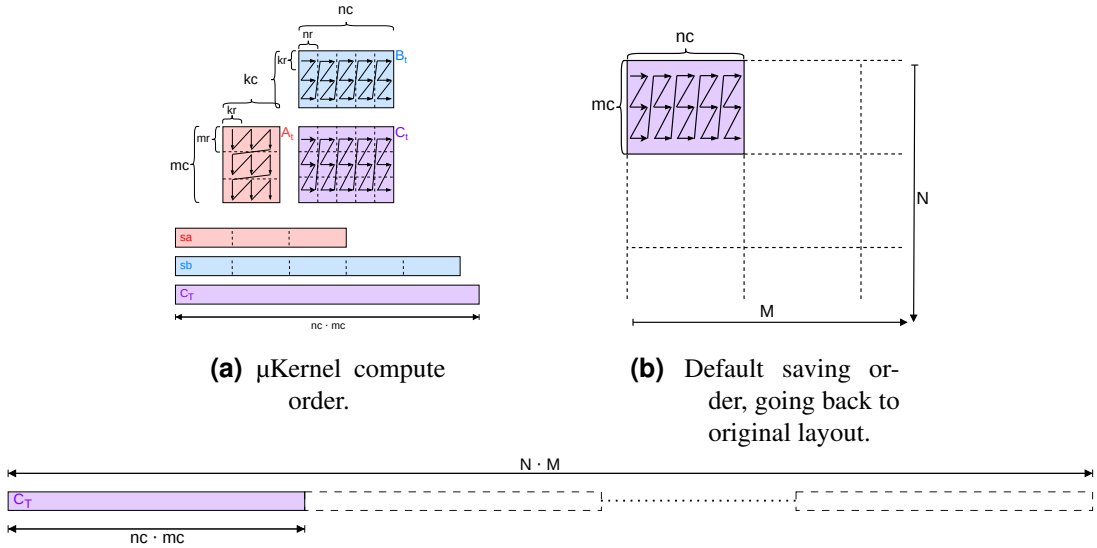


Figure 3. Overview of sequential GEMM operations using different data layouts: (left) default layout, (right) using matrix transposes. The results are the same as using OpenBLAS with column-major or row-major, respectively.



(a) µKernel compute order.

(b) Default saving order, going back to original layout.

(c) Contiguous storage of C, propagating packing layout.

Figure 4. µKernel Layout.

3.1. Kernel Implementation

LP-GEMM is built on the OpenBLAS GEMM kernel (Fig. 2c), which we extend rather than replace. This choice keeps the surrounding tiling and driver logic unchanged and isolates the impact of layout propagation from any unrelated changes to the library.

To understand why layout propagation is not supported in BLAS-style GEMMs, it is important to distinguish three concepts: (1) the *packed layout* used for reading inputs, determined by the packing strategy and specifying the layout the µKernel expects; (2) the *iteration order* used during computation, which dictates the order sub-blocks of the output are produced; and (3) the *final stored layout*, i.e., the layout in which the output matrix is written to memory and returned to the user.

BLAS constrains only the final visible output layout (3); it mandates neither the computation order (2) nor the packed layout (1), both being internal optimization details.

In most GotoBLAS-derived implementations, however, (1) and (2) are tightly coupled—the packed layout determines how the µKernel accumulates partial results, which in turn fixes the computation order—while (3) must still conform to the BLAS API and so typically differs from both, forcing an extra unpacking/reordering step (Figs. 4a and 4b).

LP-GEMM aims to unify these three aspects: making (1) = (2) = (3), so that no packing or unpacking-related reordering is necessary. By aligning the packed layout, the computation order, and the final stored layout, LP-GEMM enables the output of one GEMM to be consumed directly by the next without redundancy. Data can then be stored sequentially in memory using this common layout, improving spatial locality, as demonstrated in Fig. 4c. This design choice, however, violates the BLAS requirement that results be returned in a specific canonical layout, so LP-GEMM is not compliant with the BLAS API. In practice, the impact is bounded: the ini/mid/end-GEMM kernels are added alongside the standard BLAS interface, so existing GEMM calls remain unchanged. In-

side a propagation region, the standard GEMM calls are replaced by the new variants, and any non-GEMM operator on the intermediate matrices must be adapted to the propagated layout. This substitution is structurally mechanical, depending only on statically known chain boundaries and per-operator strides, which suggests it is a candidate for an automated compiler pass; realizing and evaluating such a pass, together with the region selection and non-GEMM operator adaptation it would require, is left for future work (Section 6). The propagated layout adds no memory overhead, as it is imposed on the original matrix.

To propagate layouts across GEMM calls without sacrificing correctness, LP-GEMM defines three kernels: the **Initial Kernel**, the **Intermediate Kernel**, and the **Ending Kernel**. These rely on two μKernels: the **Propagate-Layout μKernel**, which maintains the propagated layout across GEMMs, and the **Default μKernel**, which terminates propagation and restores the BLAS-style output layout.

Here we describe how the original kernel of Fig. 2c is modified to produce these three variants. **(1) Initial Kernel.** It starts a propagation region and must, therefore, precede every other LP-GEMM call in the sequence. It is structurally identical to the OpenBLAS kernel; the only change is that it invokes the Propagate-Layout μKernel rather than the default one, so its output is written in the LP-GEMM propagated layout. **(2) Intermediate Kernel.** It assumes the multiplier has already been packed into the layout that the μKernel expects, so it skips the input-packing step entirely; it can only be called after the multiplier has been packed, either by a preceding Initial Kernel or by an explicit packing step. Its output is written in the same order in which it is computed, continuing the propagation, which makes this kernel the workhorse for most GEMMs in a long chain. Relative to Fig. 2c, it drops lines 3–7 and starts the loop on line 8 at position 0 instead of mc. **(3) Ending Kernel.** It terminates propagation by writing its output in the canonical BLAS layout. It shares the same structural changes as the Intermediate Kernel, but invokes the Default μKernel, which itself rearranges the data back into the original layout.

3.2. μKernel Design

Both the Propagate-Layout and the Default μKernels use the same structure to calculate the values of C; the arrow in Fig. 4a shows the order in which the values of C are calculated. The default μKernel implementation, shown in Fig. 4b, has to save C in an order different from the one in which it is computed so that the resulting output has the same layout as the original operator.

The Propagate-Layout μKernel takes advantage of the fact that C is computed in a *packed* layout, which can be expressed as:

$$\frac{N}{nc}\frac{M}{mc}\frac{nc}{nr}\frac{mc}{mr}\,nr\;mr \tag{2}$$

Data is contiguous in the $nr \times mr$ dimension, which matches the micro-architecture's vector register layout. More importantly, C is also contiguous along the outer $nc \times mc$ dimension, which is the same dimension a downstream consumer GEMM will use to load it.

Because C is no longer stored in its canonical layout, two conditions must hold: (1) if C contains data before the first GEMM and $\beta \neq 0$, C must be pre-packed into

the layout of Equation 2; and (2) every μKernel call must address the regions of C as they appear in the propagated layout rather than in the original one. The propagating LP-GEMM kernels (Initial and Intermediate) already handle (2) internally.

Condition (1) requires a data reshape to the LP-GEMM layout before reaching the kernel. This same restriction is applied to most other layout propagation approaches [Liu et al. 2019, Li et al. 2024a], which insert reshapes around layout propagation regions.

3.3. Strided Loads and Stores

GEMM inputs are not always contiguous: sparse or tiled use cases require operating on strided sub-matrices, which LP-GEMM must support to remain a viable replacement for naive GEMM. OpenBLAS covers this via the standard leading-dimension parameters (`lda`, `ldb`, `ldc`) alongside M, N, and K; LP-GEMM cannot reuse this mechanism directly, since its output is written in the order in which it is computed rather than in canonical order. Instead, the kernel exposes two additional parameters: one reorders the sequence in which C_T blocks are written, and the other inserts a configurable stride between consecutive blocks, the latter making it possible to compute only a fraction of the output matrix without overwriting neighboring regions.

4. Evaluation of LP-GEMM

This section evaluates LP-GEMM on chains of three consecutive GEMM operations drawn from real Transformer, MLP, and HPC workloads. We first describe the experimental setup, then assess performance against every competing library across the full 1–24 thread range, and finally characterize how layout propagation and thread-level parallelism interact.

4.1. Experimental Setup

We evaluate LP-GEMM on an Intel Xeon Gold 6252 server processor (Table 1), pinning all runs to its 24 physical cores and leaving SMT siblings unused. We compare against OpenBLAS 0.3.34 (`SKYLAKEX` target, hand-tuned AVX-512 μKernel), Intel MKL 2026.1 [Intel Corporation 2024], oneDNN 3.13.0 [Li et al. 2024a], BLIS 2.1 [Van Zee and van de Geijn 2015], and FlashGEMM [Zhang et al. 2025] (in-tree implementation). All implementations are invoked through the BLAS Level-3 GEMM interface where possible; LP-GEMM and FlashGEMM required minor wrappers for functional equivalence.

Table 1. Execution environment.

CPU	Intel Xeon Gold 6252 (Cascade Lake)
Cores used	24 physical (SMT unused; 48 logical present)
Vector extension	AVX-512
m_c, n_c, k_c	448, 16384, 448
m_r, n_r	16, 4
L1 cache	32 KiB/core
L2 cache	1024 KiB/core (private)
L3 cache	35.8 MiB (shared)
CPU frequency	2.10 GHz (Turbo Boost disabled)
Vector register width	512 bits
Peak FMA throughput (per core, fp32, at 2.10 GHz)	134.4 GFLOPS

Two LP-GEMM configurations are reported throughout. The `lp_mixed` configuration runs the measured three-GEMM chain as Ini–Mid–End and returns its output in canonical BLAS layout, so it pays the full cost of entering and leaving a propagation region on a short chain. The `lp_mid` configuration runs every GEMM of the chain as an Intermediate kernel, paying neither the entry packing nor the exit restore. It isolates the per-GEMM cost of LP-GEMM in a sequence of GEMMs, such as the stacked blocks of a deep MLP or Transformer (with the intervening operators adapted to the propagated layout, Section 3), where the entry and exit costs are paid once and buried under many Intermediate GEMMs. The `lp_mixed` configuration is the short-chain case. Only `lp_mixed` returns output directly comparable to the row-major libraries; `lp_mid`'s output remains in the propagated layout.

4.1.1. Methodology

Each measurement is one of 20 independent single-execution runs (200 ms inter-run sleep, preceded by a 5-iteration warmup), swept over 9 thread counts $\{1, 2, 4, 6, 7, 8, 12, 16, 24\}$ and the 14 chained-GEMM workload shapes of Table 2. Turbo Boost was disabled and the governor was set to `performance`; the CPU max frequency was confirmed at 2.10 GHz during the run itself, not the platform's 3.7 GHz nominal boost clock. Each library is measured in its own OS process per thread count, removing cross-library OpenMP thread-pool contamination that a shared-process design cannot avoid under `OMP_WAIT_POLICY=active`. Every chain is verified against OpenBLAS by matrix max-norm relative error before its timing is trusted, at a per-shape threshold of $20 \cdot K_{\text{total}} \cdot \epsilon$ (K_{total} the chain's summed accumulation depth, ϵ the float32 machine epsilon); observed error was two to four orders of magnitude inside this threshold on every shape, at every thread count, with zero correctness failures. The cross-run coefficient of variation, taken over the 98 (shape, library) cells at each thread count, has a median of 0.45–0.72% across thread counts, with 1–8 of those 98 cells above 5%; we do not claim a uniform sub-5% bound.

Table 2. Matrix sizes extracted from state-of-the-art applications on HPC and AI.

Application	Domain	(M1, K1, M2, M3, N)
llama32_1b_dec	LLM-FFN	2048, 2048, 8192, 2048, 1
llama32_1b_n128	LLM-FFN	2048, 2048, 8192, 2048, 128
llama32_1b_n500	LLM-FFN	2048, 2048, 8192, 2048, 500
bert_base_n128	LLM-FFN	768, 768, 3072, 768, 128
vit_l14_n256	LLM-FFN	1024, 1024, 4096, 1024, 256
vgg16_fc_n1	DNN-MLP	4096, 25088, 4096, 1000, 1
vgg16_fc_n128	DNN-MLP	4096, 25088, 4096, 1000, 128
dlrm22_top_n4k	DNN-MLP	1024, 1024, 512, 256, 4096
dlrm22_top_n16k	DNN-MLP	1024, 1024, 512, 256, 16384
chain3_l2	HPC	400, 512, 384, 448, 384
chain3_l3	HPC	800, 1024, 768, 896, 768
chain3_dram	HPC	1600, 2048, 1536, 1792, 1536
krylov_tallsk	HPC	4096, 128, 128, 4096, 128
panel_wideshort	HPC	128, 2048, 128, 128, 2048

Table 3. Single-thread (T=1) results on the 14 workloads of Table 2. Norm. time is relative to the fastest of the five competing libraries on each workload, so values below 1.00 are wins over all of them (bold: more than 0.5% faster). GFLOP/s is absolute throughput (core peak: 134.4). L3 miss is last-level-cache miss traffic in bytes per kFLOP (64 B lines); Instr./FLOP is retired instructions per FLOP (ideal AVX-512 fp32 FMA: 0.031). The last block names the best competitor and reports its own values.

		LLM-FFN					DNN-MLP						HPC		
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
LP-GEMM (mixed)	Norm. time	4.07	1.31	1.07	1.24	1.10	3.87	1.46	1.07	1.04	1.06	1.04	1.01	1.14	1.17
	GFLOP/s	1.7	81	111	100	104	1.7	81	118	117	104	119	122	89	90
	L3 miss (B/kFLOP)	561	5.3	1.7	0.5	2.7	464	4.9	0.5	0.3	0.0	0.1	1.0	0.0	0.0
	Instr./FLOP	0.464	0.062	0.060	0.062	0.061	0.470	0.063	0.062	0.062	0.064	0.061	0.060	0.063	0.072
LP-GEMM (mid)	Norm. time	4.12	1.28	1.05	1.22	1.06	3.75	1.40	1.00	**0.97**	1.00	1.00	**0.99**	1.12	**0.85**
	GFLOP/s	1.6	83	114	101	108	1.7	85	126	126	111	124	124	90	125
	L3 miss (B/kFLOP)	532	5.2	1.6	0.4	2.6	465	4.6	0.4	0.1	0.0	0.0	0.9	0.0	0.0
	Instr./FLOP	0.472	0.062	0.059	0.062	0.060	0.471	0.062	0.059	0.059	0.060	0.059	0.059	0.063	0.059
OpenBLAS	Norm. time	2.06	1.05	1.00	1.14	1.00	1.98	1.16	1.07	1.26	1.03	1.03	1.01	1.08	1.10
	GFLOP/s	3.3	101	119	108	115	3.2	102	118	97	107	121	122	94	96
	L3 miss (B/kFLOP)	178	1.6	0.7	0.0	0.4	132	1.4	1.0	1.8	0.0	0.2	0.6	0.0	0.1
	Instr./FLOP	0.882	0.064	0.061	0.064	0.062	0.880	0.064	0.060	0.060	0.061	0.060	0.059	0.066	0.062
Best competitor	Library	MKL	MKL	OpenBLAS	MKL	OpenBLAS	MKL	MKL	Flash	Flash	MKL	Flash	MKL	MKL	Flash
	Norm. time	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	GFLOP/s	6.8	106	119	124	115	6.4	118	126	122	110	124	123	101	106
	L3 miss (B/kFLOP)	723	0.7	0.7	0.0	0.4	833	0.4	0.1	1.0	0.0	0.0	0.6	0.0	0.0
	Instr./FLOP	0.044	0.056	0.061	0.056	0.062	0.042	0.056	0.056	0.056	0.050	0.056	0.049	0.056	0.057

4.2. Consecutive GEMMs

We evaluate LP-GEMM in its intended setting: a chain of three sequential GEMMs where the output of each call feeds the next. Matrix sizes (Table 2) are drawn from production workloads spanning three categories. **Transformer FFN blocks (LLM-FFN)** model the $O_{\text{proj}} \rightarrow \text{FFN}_{\text{up}} \rightarrow \text{FFN}_{\text{down}}$ chain inside one transformer block when projections are not kernel-fused: LLaMA 3.2-1B [Meta's Llama 3 team 2024] (embedding 2048, FFN 8192) at decode (N=1) and two prefill lengths, BERT-base [Devlin et al. 2019] (768/3072), and ViT-L/14 [Dosovitskiy et al. 2021] (1024/4096). **Classical DNN MLPs (DNN-MLP)** represent fully-connected stacks: VGG-16's classifier head [Simonyan and Zisserman 2015] at batch 1 and 128, and three GEMMs from the 4-FC top MLP of DLRM-2022 [Naumov et al. 2019] at two batch sizes. **HPC matrix-chain products** comprise three non-square chains sized to fit in L2, fit in L3, and exceed L3 (Polybench [Pouchet and Yuki 2016]-style 3mm dimensions), plus two shape-extreme chains modeling Krylov-style tall-skinny iterations and LAPACK-style wide-short panel updates [Golub and Van Loan 2013]. The dataset spans N from 1 to 16,384 and arithmetic intensities from L2-resident to DRAM-bound. We also implemented LP-GEMM for RISC-V (RVV); results are reported for x86 only, since MKL, oneDNN, and FlashGEMM have no RISC-V support and only OpenBLAS and BLIS could be compared there.

Single-threaded, LP-GEMM matches the field on compute-bound workloads. The `lp_mid` configuration is within 7% of the best competitor on 8 of the 14 workloads, all among the 12 compute-bound ones (Table 3). It beats it by more than 0.5% on three (#14 by 15%, #9 by 3%, #12 by 1%) and ties it on three more (#8, #10, #11), sustaining 92–94% of the core's peak on five of the six. The comparison that isolates propagation is against OpenBLAS, whose kernel structure LP-GEMM keeps: `lp_mid` is faster on those same six workloads, by up to 23% (#9, #14), and retires no more instructions per FLOP than OpenBLAS on all twelve compute-bound workloads (strictly fewer on 11),

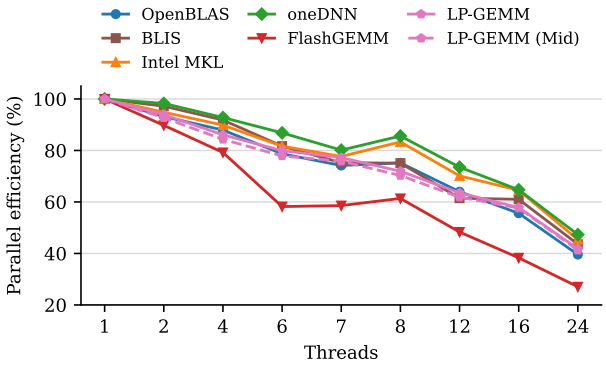


Figure 5. Geomean parallel efficiency (speedup relative to each library's own T=1 time, divided by T) across the 1–24 thread range for every library.

the signature of the removed pack/unpack work.

The losses are concentrated where there is little packing to remove. At N=1 (#1, #6) every library is memory-bound at 1–7 GFLOP/s, and LP-GEMM is about 2× slower than OpenBLAS and 4× slower than MKL. The counters place the cause in memory traffic and not in instruction overhead: LP-GEMM retires half the instructions per FLOP of OpenBLAS (0.47 vs. 0.88) but incurs 3× its L3-miss traffic (e.g., 532 vs. 178 B/kFLOP on #1), whereas MKL streams the weights with 0.04 instructions per FLOP. A matrix-vector variant of the Intermediate kernel is the natural remedy. A milder group (#2–#5, #7) trails OpenBLAS by 5–22% with the same signature (fewer instructions but more L3 traffic), which suggests that when the activations are small next to the weights, propagation saves little and the extra traffic dominates. The counters are consistent with this explanation but do not isolate it.

Restoring the canonical layout is cheap on these chains. The `lp_mixed` configuration pays for it once per three GEMMs: a geomean 5.4% over `lp_mid` (at most 7.2% on 13 of the 14 workloads, 38.5% on #14).

4.3. Thread Scaling

We repeat the 14 workloads at 9 thread counts from 1 to 24 to test whether the benefit of layout propagation is a single-thread artifact. **LP-GEMM scales like OpenBLAS.** Fig. 5 shows parallel efficiency (speedup over each library's own T=1 time, divided by T): at T=24 it is 41.8%/41.3% for `lp_mixed`/`lp_mid` against 39.6% for OpenBLAS, behind MKL and oneDNN (45.4%/47.3%). Because efficiency is normalized to each library's own serial time, it describes scaling shape and not competitiveness.

Propagation and parallelism do not interact. The `lp_mid`/OpenBLAS geomean time ratio stays in a flat 1.05–1.17× band over all 14 workloads (0.99–1.09× on the 12 compute-bound ones), with no trend in thread count; `lp_mixed` tracks it 3–6 points higher, consistent with the roughly constant unpack cost.

Against the field, the compute-bound picture holds at scale. On the 12 compute-bound workloads, `lp_mid`'s geomean time relative to the best competitor goes from 1.07× at T=1 to 1.38× at T=24, against 1.07× to 1.35× for OpenBLAS; MKL (1.02–1.10×) and oneDNN (1.17–1.22×) stay flat. Clear wins over the best competi-

tor fall from 3/14 at $T{=}1$ to 2/14 at $T{=}2$, 1/14 at $T{=}4$ and none from $T{=}6$ onward, while wins over OpenBLAS alone remain between 2 and 6 of 14 at every thread count. The `lp_mixed` configuration uses $1.38\times/1.59\times/1.51\times$ the package energy of the most energy-efficient competitor at $T{=}1/T{=}8/T{=}24$, consistent with its runtime.

Five workloads (#1, #5, #10, #13, #14) plateau or regress past 16 threads under `lp_mixed`. All are cache-resident, small-batch/decode-phase or shape-extreme chains, where per-thread work shrinks below the fixed cost of launching and joining 24 threads.

5. Related Work

Traditional GEMM optimization treats each call as an isolated computation. BLAS implementations derived from GotoBLAS [Goto and van de Geijn 2008] pack every invocation's inputs and unpack its output regardless of surrounding context. This is not for lack of pipeline awareness: Intel MKL's packed API (`cblas_?gemm_pack/_compute`, available since 2017) already amortizes *input*-packing cost across repeated calls that reuse an operand, and neither MKL nor BLIS is naive about repeated computation. What no BLAS-level implementation does is write a GEMM's *output* in a non-canonical, packed layout for a downstream consumer to read directly. Every call still restores the standard row/column-major layout before returning, which is exactly the redundant work that LP-GEMM targets.

Layout propagation across operators has been studied at the graph level, where oneDNN [Li et al. 2024a] is the closest prior art: it infers blocked formats for downstream primitives and propagates them across a computation graph when enabled. NeoCPU [Liu et al. 2019] takes a similar analysis-and-search approach targeted specifically at ML graphs, trading search cost against layout quality. Both operate at the graph/primitive-framework level and require the user to mark reorder boundaries with explicit primitives. LP-GEMM addresses the same problem one layer down, inside a plain BLAS-style GEMM call in raw C rather than a graph or primitive framework: the propagation is integrated into the kernel itself, at the cost of requiring the programmer (or, per Section 3, a future compiler pass) to identify propagation regions rather than relying on graph-level inference.

FlashGEMM [Zhang et al. 2025] attacks the sequential-GEMM problem through aggressive kernel fusion, achieving layout propagation as a side effect of fusing consecutive operations and including a profitability check on memory pressure. Fusion yields strong locality on pure GEMM chains but does not accommodate intermediate non-GEMM layers without re-implementing them inside the fused kernel, a constraint LP-GEMM avoids by keeping each GEMM a discrete, individually callable API call.

6. Conclusion and Future Work

This work introduced LP-GEMM, which propagates the packing layout across a chain of dependent GEMMs via Initial, Intermediate, and Ending kernel variants, avoiding the pack/unpack cycle BLAS otherwise repeats at every chain boundary, while keeping the OpenBLAS structure. LP-GEMM is within 7% of the fastest of five libraries on 8 of the 14 workloads, beats it on three (by up to 15%, on `panel_wideshort`), and beats OpenBLAS on six (by up to 23%), excluding the layout conversions at the ends of the

chain. With more threads it follows OpenBLAS's scaling, at a flat 1.05–1.17× cost relative to it. While MKL's packed API amortizes input-packing cost across repeated calls, and oneDNN propagates layout across a computation graph, neither writes a GEMM's output in a non-canonical layout for a downstream call to consume directly. Moreover, oneDNN's propagation requires the user to operate within its graph/primitive framework. LP-GEMM closes this gap one layer down, inside a plain BLAS-style GEMM call, with no graph-compiler dependency.

Benefits and limitations. LP-GEMM removes the pack/unpack work between chained GEMMs without a graph compiler or kernel fusion, and is up to 23% faster than OpenBLAS at one thread, where packing matters. It is not a drop-in replacement: it adds parameters for strided accesses (Section 3.3), returns outputs in a propagated layout the consumer must be aware of, and needs a Propagate-Layout μKernel written with hardware intrinsics for each target (implemented for AVX-512 x86 and RISC-V RVV). It saves little where there is little packing to remove, notably at $N{=}1$, where it is about $2\times$ slower than OpenBLAS because of extra L3 traffic, and restoring the canonical layout costs a geomean 5.4% per three-GEMM chain. The evaluation covers one AVX-512 x86 CPU and single-precision chains; other CPU architectures and accelerators are not evaluated in this paper.

Future work. The central direction is automating the substitution: a compiler pass that detects GEMM chains and inserts the appropriate ini/mid/end variants, together with adapting non-GEMM operators to consume the propagated layout. A matrix-vector variant of the Intermediate kernel would address the $N{=}1$ losses, and a pack/compute/unpack breakdown would quantify what propagation removes on each shape. We also plan to evaluate the RISC-V implementation and other CPU architectures.

Acknowledgment

This research was supported by BTG Pactual, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, São Paulo Research Foundation (FAPESP) (2023/03328-9).

References

Alman, J., Duan, R., Vassilevska Williams, V., Xu, Y., Xu, Z., and Zhou, R. (2025). More asymmetry yields faster matrix multiplication. In Azar, Y. and Panigrahi, D., editors, *SODA'2025*, pages 2005–2039.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT'2019*.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale. In *9th International Conference on Learning Representations, ICLR*.

Genc, H., Kim, S., Amid, A., Haj-Ali, A., Iyer, V., Prakash, P., Zhao, J., Grubb, D., Liew, H., Mao, H., Ou, A. J., Schmidt, C., Steffl, S., Wright, J. C., Stoica, I., Ragan-Kelley, J., Asanovic, K., Nikolic, B., and Shao, Y. S. (2021). Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *DAC'2021*.

Golub, G. H. and Van Loan, C. F. (2013). *Matrix Computations*. Johns Hopkins University Press, 4th edition.

Goto, K. and van de Geijn, R. A. (2008). Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34:1–25.

Intel Corporation (2024). Intel math kernel library (Intel MKL). `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html`. Accessed: 2025-12-10.

Kim, H., Ye, G., Wang, N., Yazdanbakhsh, A., and Kim, N. S. (2024). Exploiting Intel advanced matrix extensions (AMX) for large language model inference. *IEEE Comput. Archit. Lett.*, 23(1):117–120.

Lam, M. D., Rothberg, E. E., and Wolf, M. E. (1991). The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74.

Li, J., Qin, Z., Mei, Y., Cui, J., Song, Y., Chen, C., Zhang, Y., Du, L., Cheng, X., Jin, B., Zhang, Y., Ye, J., Lin, E., and Lavery, D. (2024a). oneDNN graph compiler: A hybrid approach for high-performance deep learning compilation. In Grosser, T., Dubach, C., Steuwer, M., Xue, J., Ottoni, G., and Pereira, F. M. Q., editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2024, Edinburgh, United Kingdom, March 2-6, 2024*, pages 460–470. IEEE.

Li, M., Liu, C., Liao, J., Zheng, X., Yang, H., Sun, R., Xu, J., Gan, L., Yang, G., Luan, Z., and Qian, D. (2024b). Towards optimized tensor code generation for deep learning on sunway many-core processor. *Frontiers Comput. Sci.*, 18:182101.

Liu, Y., Wang, Y., Yu, R., Li, M., Sharma, V., and Wang, Y. (2019). Optimizing CNN model inference on CPUs. In *USENIX ATC'19*, pages 1025–1040.

Meta's Llama 3 team (2024). The Llama 3 herd of models. *arXiv e-prints*.

Naumov, M., Mudigere, D., et al. (2019). Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091.

NVIDIA Corporation (2024). cuBLAS: CUDA basic linear algebra subroutines. `https://developer.nvidia.com/cublas`. Accessed: 2025-12-10.

Pouchet, L.-N. and Yuki, T. (2016). PolyBench/C 4.2: The polyhedral benchmark suite. `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`.

Rohwedder, C. S., Henderson, N., De Carvalho, J. P. L., Chen, Y., and Amaral, J. N. (2023). To pack or not to pack: A generalized packing analysis and transformation. In *CGO'23*, page 14–27.

Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR*.

The Netlib (2025). BLAS (Basic Linear Algebra Subprograms). `https://www.netlib.org/blas/`. Accessed: 2025-11-25.

Van Zee, F. G. and van de Geijn, R. A. (2015). BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *NeurIPS'17*.

Wolfe, M. J., Shanklin, C., and Ortega, L. (1995). *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc.

Zhang, J., Yang, W., Fang, J., Dong, D., and Chen, X. (2025). FlashGEMM: Optimizing sequences of matrix multiplication by exploiting data reuse on cpus. *ACM Trans. Archit. Code Optim.*, 22(4).