

Analyzing Latency Hiding and Parallelism in an MLIR-based AI Kernel Compiler

Javed Absar¹, Samarth Narang², and Muthu Baskaran²

¹Qualcomm Technologies International, Ltd.

²Qualcomm Technologies, Inc.

{mabsar, samanara, muthub}@qti.qualcomm.com

Abstract

AI kernel compilation for edge devices depends on the compiler’s ability to exploit parallelism and hide memory latency in the presence of hierarchical memory and explicit data movement, among other factors. This paper reports a benchmark methodology and corresponding results for three compiler-controlled mechanisms in an MLIR-based compilation pipeline: vectorization (Vec), multi-threading (MT) across hardware contexts, and double buffering (DB) using ping-pong scratchpad buffers to overlap DMA transfers with compute. Using Triton/Inductor-generated kernels, we present an ablation ladder that separates the contribution of Vec, MT, and DB, and we quantify how MT speedup scales with problem size using GELU as a representative activation kernel. The results show that vectorization provides the primary gain for bandwidth-sensitive kernels, MT delivers substantial improvements once scheduling overhead is amortized, and DB provides additional benefit when transfers and compute can be overlapped (i.e., outside the extremes of purely memory-bound or purely compute-bound behavior).

1 Introduction

Kernel optimization is difficult to automate: hand-written kernels remain hard to beat, yet production systems require portable and maintainable code generation across rapidly evolving architectures. On edge NPUs, performance is shaped by hierarchical memory, explicit DMA-managed transfers, and the need to schedule work so compute stays busy while transfers are in flight.

End-to-end model results are essential, but they often obscure *which* mapping mechanisms improve performance and *why*. We therefore adopt a reproducible, kernel-oriented evaluation methodology aligned with Triton/Inductor-style code generation, benchmarking representative operator kernels under controlled problem sizes to separately quantify the impact of vectorization (Vec), multi-threading (MT), and double buffering (DB) [1]. Vec exploits data-level parallelism; MT exploits loop- and region-level parallelism by distributing independent tiles across hardware contexts; and DB reduces stall time by overlapping memory transfers with compute.

To attribute gains to specific mechanisms, we report results using a simple ablation ladder:

$$\text{Scalar} \rightarrow \text{Vec} \rightarrow \text{Vec+MT} \rightarrow \text{Vec+MT+DB}.$$

In this ladder, Vec isolates SIMD-style lowering, Vec+MT quantifies incremental thread-level speedup, and Vec+MT+DB evaluates whether an explicit latency-hiding schedule provides additional improvement once Vec and MT are already in place. In addition to reporting results, we describe the concrete MLIR IR patterns and pass structure used to realize MT and DB, making the methodology easy to reproduce and extend.

2 Implementation Details: Multi-threading and Double Buffering

Our implementation is built in MLIR [2] and follows a design principle: keep the intent expressed in structured IR for as long as possible, and lower to runtime constructs only after the compiler has imposed a schedule. This improves portability and makes transformations easier to validate.

2.1 Multi-threading (MT)

Hardware model. The target NPU supports multi-threaded vector execution, where each hardware thread is associated with an independent vector context (e.g., vector register and predicate state). This enables concurrent execution of independent tiles of the same kernel, provided the tiled iteration space can be partitioned without cross-thread dependences.

Two-stage MT lowering. We implement MT as a two-stage pipeline that preserves structured parallelism and then introduces an explicit fork-join. First, *Form-Virtual-Threads* rewrites a tiled kernel (e.g., `linalg.generic`) into an explicitly parallel form using `scf.forall`. The pass uses a size-based profitability heuristic over the tile space and selects a distribution policy (block vs. block-cyclic) to balance work when ranges are uneven.

Second, *Form-Async-Threads* lowers `scf.forall` to a fork-join representation using MLIR’s Async [3]. Each tile becomes an `async.execute` region that produces a token; tokens are collected into an `async` group, and `async.await_all` forms a barrier before subsequent dependent computation.

Canonical fork-join skeleton in IR. The generated pattern is intentionally small and regular, which makes it straightforward to lower into a coroutine/task runtime [3]:

```
%group = async.create_group %N
scf.for %tile = ... {
  %tok = async.execute { /* tile body */ async.yield }
  async.add_to_group %tok, %group
}
async.await_all %group
```

Why Async (instead of lowering MT directly). Keeping MT as a structured fork-join in Async preserves parallel semantics in a declarative form until late lowering, while still enabling a straightforward translation to runtime scheduling [3].

2.2 Double buffering (DB)

Double buffering is a software-pipelining strategy that overlaps transfers and compute by alternating between two scratchpad buffers (ping and pong). The technique is closely related to modulo scheduling and pipelined loop execution [4, 5]. We implement DB in two stages, separating the construction of a pipelined schedule from the introduction of target-specific asynchronous transport primitives.

Stage 1: structural pipelining. Stage 1 matches a single-buffered tiled-loop “normal form” typically created by tiling:

```
memref.subview → memref.alloc → memref.copy,
```

followed by compute and then a write-back sequence. From this structure, the pass builds an explicit ping-pong schedule. It emits a prologue that prefetches the first tile into ping buffers, then rebuilds the main loop into two alternating sub-kernels. Each sub-kernel (i) issues a prefetch of the *next* tile into the opposite buffer, (ii) computes using the current buffer, and (iii) reconstructs storeback by rematerializing subviews at the current induction variable. A boolean toggle selects ping vs. pong each iteration, and the pass attaches lightweight attributes as anchors so the next stage can reliably identify prefetch/compute/storeback regions.

Stage 2: asynchronous DMA integration. Stage 2 replaces synchronous copies with explicit asynchronous DMA operations. Prefetch paths are rewritten to `memref.dma_start` with distinct ping/pong tags, and `memref.dma_wait` is inserted immediately before compute to ensure tile residency in TCM. Storeback copies can be handled similarly, and the pass emits balanced tag deallocations. By staging the transformation, we first establish a correct schedule and only then map it onto the target’s transport interface, mirroring classic compiler practice for pipelined loops [4, 5].

Composing DB with MT. DB composes naturally with MT. Once `memref.dma_wait` enforces that a tile is resident in TCM, the compute region can exploit `scf.forall` (virtual threads) and/or the lowered fork-join representation to execute multiple sub-tiles concurrently. This is the composition used to realize the Vec+MT+DB rung of our ladder.

3 Benchmark Setup and Methodology

We evaluate two representative kernels. The first is a bandwidth-centric 2D vector addition microbenchmark with shape $[64, 128 \times 128]$ elements, which is useful for understanding the relative impact of vectorization and latency-hiding in a memory-heavy regime. The second is GELU, a common activation kernel representative of transformer inference subgraphs; we use a Triton implementation of GELU as our concrete kernel instance.

For the vector-add microbenchmark, we evaluate the ladder variants *Scalar*, *Vec*, *Vec+MT*, and *Vec+MT+DB*. For GELU, we report end-to-end latency across a problem-size sweep for both single-threaded and multi-threaded execution to expose MT overhead amortization and scaling. We report latency in microseconds (μ s) and, where applicable, speedup as the ratio of single-thread to multi-thread time.

4 Results

4.1 vec-add-2d Ablation Ladder

Figure 1 summarizes the ladder for vec-add-2d. Vectorization accounts for the dominant improvement ($132,479 \mu\text{s} \rightarrow 3,210 \mu\text{s}$, $\sim 41.3\times$), consistent with a bandwidth-oriented kernel that benefits immediately from SIMD-style lowering. MT and DB provide smaller but measurable incremental gains ($3,210 \mu\text{s} \rightarrow 3,000 \mu\text{s} \rightarrow 2,689 \mu\text{s}$), suggesting that once vectorization is in place, remaining headroom comes from reducing synchronization overheads and partially overlapping transfer/compute effects rather than from increasing arithmetic throughput.

4.2 GELU: Single-thread vs Multi-thread Scaling

Figure 2 reports GELU latency over a size sweep and Figure 3 shows the corresponding multi-thread speedup. MT improves performance across all tested sizes and the speedup grows with problem size, reaching $\sim 3.91\times$ at 1,048,576 elements ($12,947 \mu\text{s}$ single-thread vs $3,313 \mu\text{s}$ multi-thread). This trend indicates that fixed

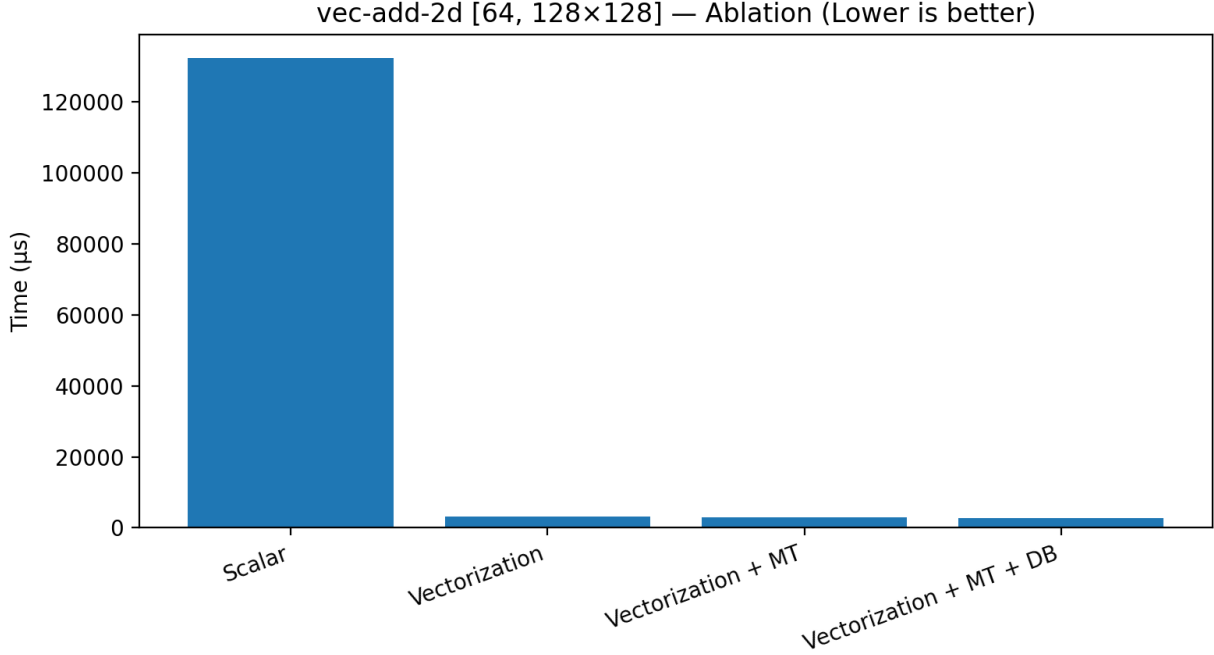


Figure 1: vec-add-2d ($[64, 128 \times 128]$ elements) ablation ladder: Scalar (132,479 μs), Vec (3,210 μs), Vec+MT (3,000 μs), Vec+MT+DB (2,689 μs).

overheads associated with fork–join scheduling are amortized as per-thread work increases, while saturation at larger sizes suggests emerging shared bottlenecks such as memory bandwidth or barrier costs.

4.3 Discussion

Across these kernels, vectorization provides the key first-order win, especially in bandwidth-sensitive regimes. MT then yields substantial additional improvement when there is sufficient parallel slack and the kernel is large enough to amortize scheduling overhead. DB provides incremental benefit when transfers and compute are both significant; in the purely memory-bound limit, DB is constrained by transfer bandwidth, and in the purely compute-bound limit, overlap opportunities are limited.

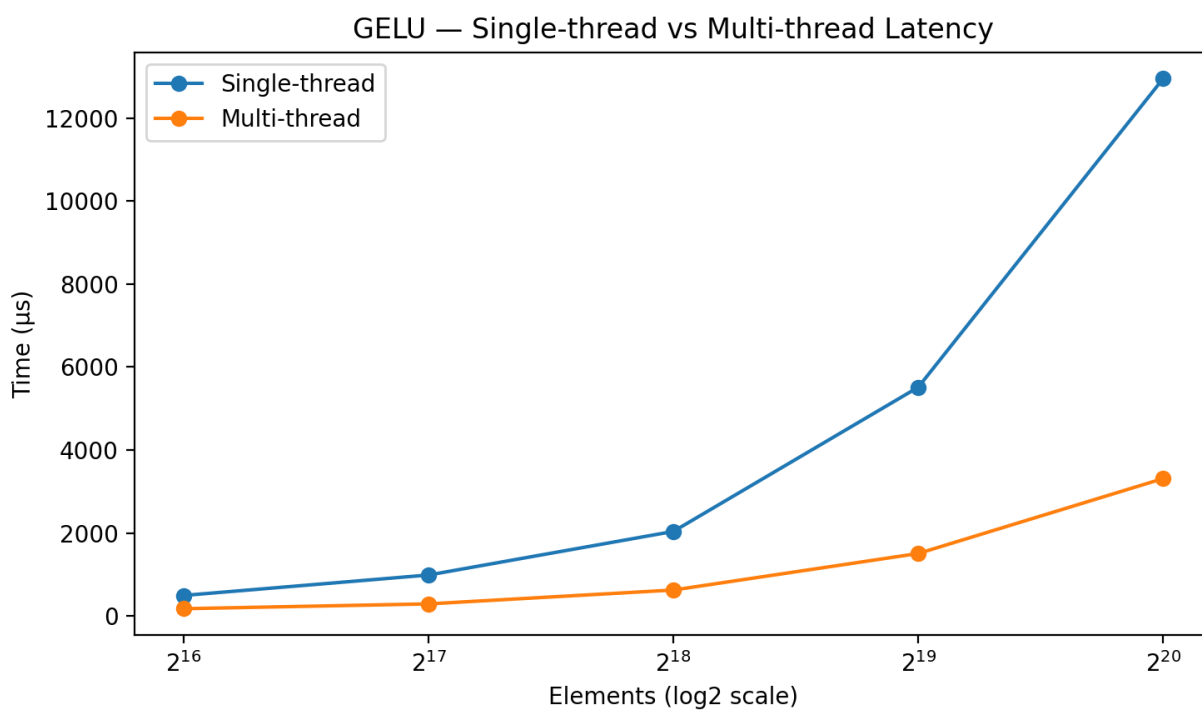


Figure 2: GELU latency vs problem size for single-threaded and multi-threaded execution.

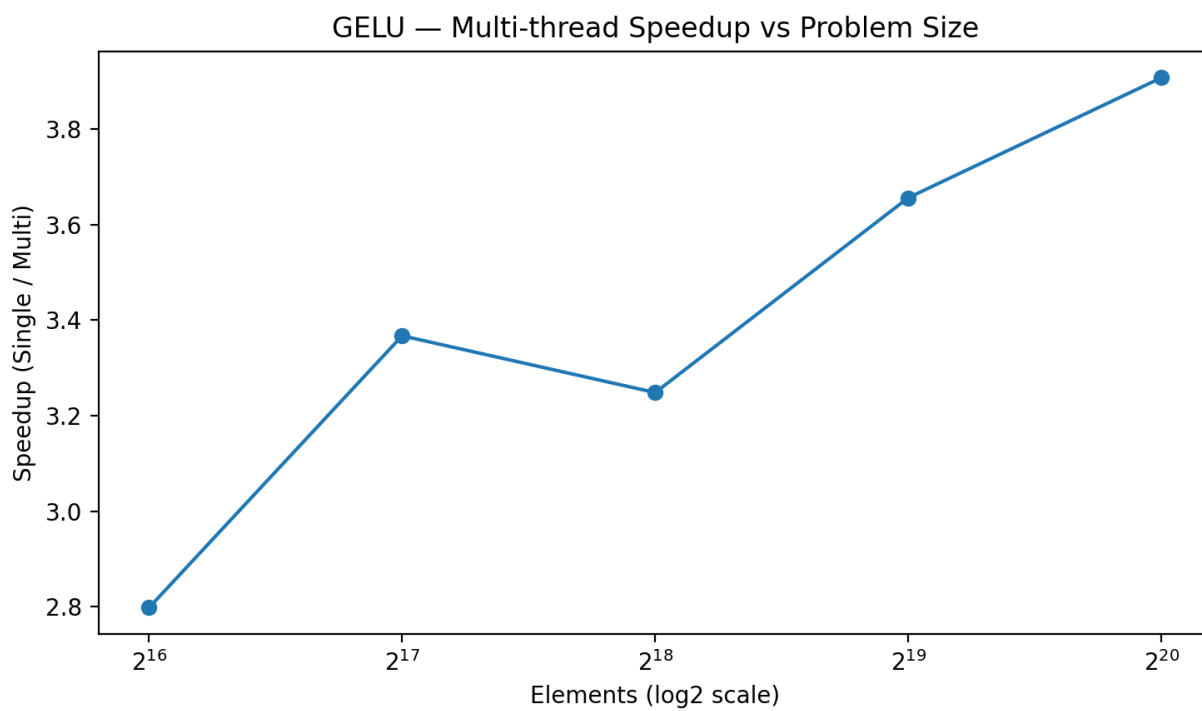


Figure 3: GELU multi-thread speedup (Single/Multi) vs problem size.

5 Conclusion

We presented a compact, reproducible methodology for analyzing vectorization, multi-threading, and double buffering in an MLIR-based kernel compiler. The proposed ladder makes it easier to attribute performance improvements to specific compiler mechanisms rather than reporting only end-to-end speedups. Our measurements show that vectorization is foundational, MT can provide large gains once overhead is amortized, and DB adds incremental benefit when transfer/compute overlap exists. Future work will broaden the benchmark set (e.g., RMSNorm, softmax) and connect these measurements to a predictive model that relates overlap efficiency to DMA throughput and scratchpad capacity.

References

- [1] Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, 2019.
- [2] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore’s law, 2020.
- [3] LLVM Project. Mlir async dialect. <https://mlir.llvm.org/docs/Dialects/AsyncDialect/>, 2026. Accessed 2026-01-21.
- [4] Monica S. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI)*, 1988.
- [5] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO)*, 1994.