GPU-Accelerated Low-Latency Inference Engine for Market Signals

Custom CUDA Kernels, CUDA Graphs, and Streaming-Oriented Memory Optimizations

Mail Mamedov

January 5, 2026

Abstract

The objective of this assignment is to design and evaluate a GPU-accelerated inference engine for low-latency market-signal prediction, targeting batch size 1 and microsecond-scale response time. Because small models are dominated by kernel-launch and data-movement overhead rather than raw throughput, the solution explores (i) a cuBLAS-based baseline pipeline, (ii) custom dense-layer CUDA kernels that fuse matrix–vector multiplication (GEMV), bias, and activation, and (iii) CUDA Graph replay to reduce repeated dispatch overhead.

On an NVIDIA Tesla T4, CUDA Graphs reduced the cuBLAS pipeline median kernel-only latency from 24.384 µs to 12.288 µs. A custom warp-per-output kernel (one warp computes one output neuron) achieved 13.952 µs (direct) and 8.448 µs (CUDA Graph) median kernel-only latency with improved tail latency. An initial "thread-per-output" kernel based on a prior MByV design was initially much slower (106.784 µs p50) due to extra weight staging and unfavorable memory access patterns; by changing only the weight storage format to a packed K-major layout (without changing the thread mapping), the same algorithm improved to 18.848 µs p50 under CUDA Graph replay.

End-to-end experiments including H2D/D2H transfers showed that pinned host memory and CUDA Graph replay provided the best end-to-end configuration at 21.632 µs p50 and 28.288 µs p99. A persistent-kernel, zero-copy streaming design eliminated per-tick launches but was slower on PCIe-attached memory (52.659 µs p50). Overall, the results demonstrate that for batch-1 inference, careful kernel mapping, memory layout, and CUDA Graph replay are decisive for reducing both median latency and jitter.

1 Design Methodology

This section describes the algorithm being accelerated, where parallelism is introduced, and how the CPU (master) and GPU (slave) cooperate to execute low-latency inference.

1.1 Algorithm overview

We consider a compact feed-forward neural network used for market-signal prediction, executed at batch size 1. Each dense layer computes:

$$\mathbf{y} = \phi(\mathbf{W}\mathbf{x} + \mathbf{b}), \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^K$ is the input feature vector, $\mathbf{W} \in \mathbb{R}^{M \times K}$ is the weight matrix, $\mathbf{b} \in \mathbb{R}^M$ is the bias, and $\phi(\cdot)$ is an activation function (ReLU in this work, and identity for the output layer). The project uses a fixed-size three-layer network:

$$512 \to 256 \to 64 \to 1, \tag{2}$$

representative of small "tick" models that must run within tens of microseconds.

The computational core is matrix–vector multiplication (GEMV). For a single output neuron i:

$$y_i = \phi\Big(\sum_{j=0}^{K-1} W_{ij} x_j + b_i\Big). \tag{3}$$

1.2 Where parallelism is introduced

Batch size 1 eliminates parallelism across samples; therefore, the implementation exploits parallelism across output neurons and, when possible, across the dot-product reduction.

Three execution strategies are considered:

1. **cuBLAS baseline pipeline.** Each layer uses a GEMM call with $N = 1$ (equivalent to GEMV), with separate kernels for bias and activation.

2. **Warp-per-output dense layer.** Each warp computes one output neuron (one row of $\mathbf{W}$). Lanes compute partial sums over strided subsets of j and reduce with warp intrinsics. Bias and activation are fused in the same kernel.

3. **Thread-per-output (MByV-style) dense layer.** Each thread computes one output neuron y_i. The input vector is broadcast within a warp using shuffle intrinsics so that at step j, each lane uses the same x_j while reading its own weight W_{ij}. Bias and activation are fused. This mapping is intentionally preserved to match the original algorithm; optimization focuses on memory layout rather than thread mapping.

1.3 Master and slave responsibilities

Master (CPU host code). The host is responsible for:

- allocating device buffers and (for end-to-end tests) host buffers,

- transferring inputs to the GPU and outputs back to the host (synchronous or asynchronous),

- launching kernels (direct launches or CUDA Graph replay),
- collecting latency measurements and validating correctness against a CPU reference.

Slaves (GPU kernels / cuBLAS kernels). The GPU executes:

- dense-layer compute (custom kernels or cuBLAS GEMM kernels),
- fused bias and activation operations (custom kernels) or separate epilogue kernels (baseline),
- an optional persistent-kernel loop for streaming input.

1.4 Communication and synchronization

The master and slaves communicate via CUDA global memory and the CUDA runtime:

- **Standard mode:** host-to-device (H2D) copy for input features, kernel(s) on device-resident buffers, and device-to-host (D2H) copy for outputs.
- **Pinned-memory mode:** page-locked host buffers enable faster/more predictable DMA and support asynchronous copies.
- **CUDA Graph mode:** the master records a fixed sequence of operations (kernels and optional copies) into a graph and replays it using `cudaGraphLaunch`, reducing CPU dispatch overhead.
- **Persistent-kernel mode:** host and device share a mapped pinned buffer. The host publishes new inputs by incrementing a sequence counter; the GPU polls and processes ticks, then publishes completion. This eliminates per-tick launches but may incur PCIe latency when reading mapped host memory.

1.5 Warp-per-output kernel structure

Figure 1 illustrates the warp-per-output mapping. The input vector $\mathbf{x}$ is staged into shared memory (small and reused). Each warp loads weights for one output row, computes partial products, reduces within the warp, applies bias and activation, and writes the output.

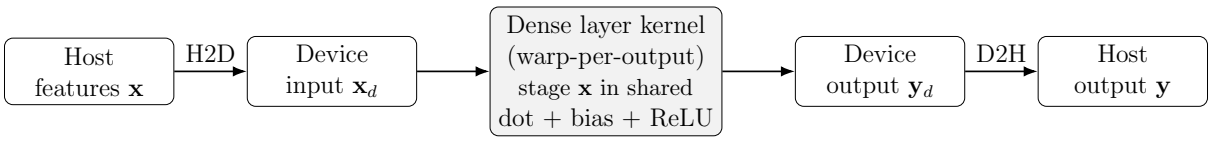

Figure 1: Dataflow for one dense layer. The custom kernel assigns one warp to one output neuron and fuses dot product, bias, and activation.

1.6 Thread-per-output (MByV-style) and packed weight layout

In the thread-per-output approach, a warp processes multiple output neurons concurrently (one per lane). At dot-product step j, all lanes need the same feature x_j. Instead of loading x_j separately per thread, a single lane loads x_j and broadcasts it via `__shfl_sync`.

The dominant bottleneck is weight access. With conventional row-major storage (W_{ij} contiguous in j for fixed i), a warp reading $W_{i\ell,j}$ for $\ell = 0..31$ accesses memory with a stride of K between lanes, which leads to non-coalesced global loads. To preserve the original algorithm while improving memory efficiency, the weights are packed once (offline) into a K-major layout:

$$W_{packed}[j \cdot M_{pad} + i] = W_{rm}[i \cdot K + j], \tag{4}$$

where M_{pad} is M rounded up to a multiple of 32. This makes the warp's weight loads at fixed j contiguous across lanes, improving coalescing. Figure 2 contrasts the two layouts.

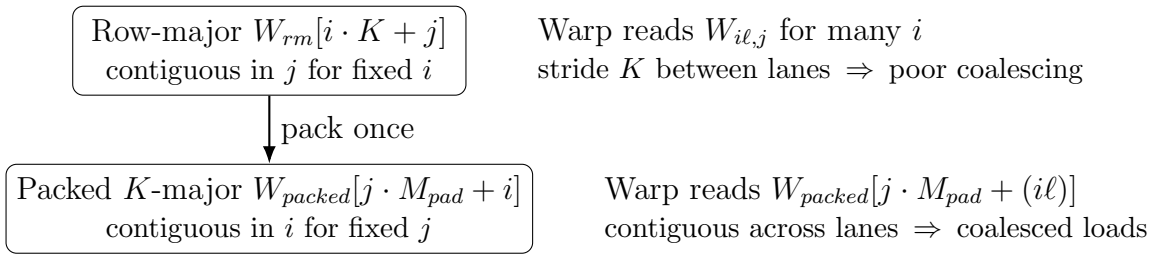

Figure 2: Weight packing to improve coalescing for the thread-per-output (MByV-style) mapping without changing the mapping itself.

Structured boundary handling using power-of-two tiles. A distinctive and technically interesting component of the original kernel is its treatment of *edge regions* when dimensions are not multiples of the warp size and/or the chosen tiling granularity. Instead of falling back to scalar cleanup code, the kernel covers the remainder by a compile-time generated schedule of rectangular tiles whose row counts are powers of two and whose column counts are either (i) multiples of 32 (the "regular" stripe) or (ii) a small final remainder.

Row remainder. After processing the largest multiple of 32 rows in a block, the remaining row count is $r \in [1, 31]$. The kernel decomposes r into a sum of powers of two, $r = \sum_t 2^{p_t}$, and processes each group of $n = 2^{p_t}$ rows as a tile. For a tile of n rows, the warp is partitioned into n "row lanes" and $32/n$ cooperating lanes per row. Each cooperating lane accumulates a disjoint subset of the feature slice, and a fixed-pattern shuffle reduction sums these partial results across the $32/n$ lanes. Because n is a power of two, the reduction pattern is regular, branch-free, and fully determined at compile time.

Column remainder. Independently, each warp's feature slice is split into a regular part whose width is a multiple of 32 and a final remainder $c \in [1, 31]$. The regular part uses a full 32-step shuffle-broadcast loop, while the remainder uses a shortened loop and treats out-of-range weights/features as zeros. When both row and column remainders are present, the remaining columns are assigned to the same power-of-two row tiles using a small precomputed schedule, so that the "corner" region is still covered by uniform rectangular tiles rather than irregular control flow.

Example: how a 597×293 layer is tiled. Figure 3 illustrates the decomposition for a representative irregular size. Consider $M = 597$ rows and $K = 293$ columns with $B = 4$ blocks and $W = 4$ warps per block. Then each block covers $\lceil 597/4 \rceil = 150$ rows and each warp covers $\lceil 293/4 \rceil = 74$ columns, implying padded dimensions $M_{\text{pad}} = 600$ and $K_{\text{pad}} = 296$. Inside a block, 128 rows form the regular region (a multiple of 32), leaving a row remainder $r = 22$, which is decomposed as $22 = 16 + 4 + 2$. Along the column dimension, each warp processes a regular stripe of width 64 and a tail of width 10. The resulting edge handling therefore covers the remainder with a small set of power-of-two row tiles crossed with either the regular stripe or the final column tail.

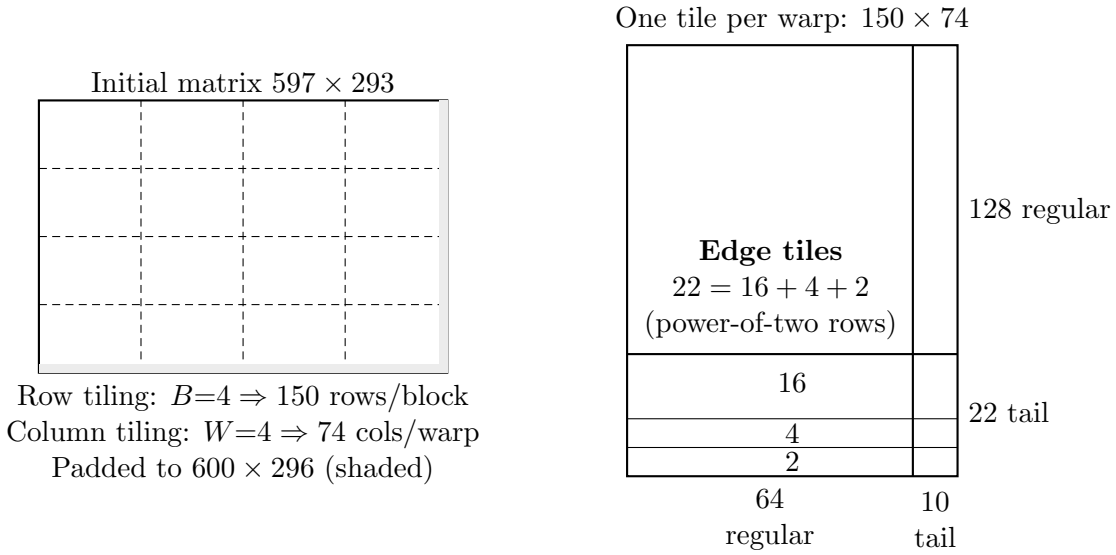

Figure 3: Schematic tiling for an irregular dense layer under the thread-per-output mapping. Left: the 597×293 matrix is covered by a 4×4 grid of tiles, with small padded bands to reach multiples of the tiling granularity. Right: a single 150×74 warp tile is split into a regular region (128×64) plus structured edge tiles. The row remainder is decomposed into power-of-two groups (16, 4, 2), enabling fixed-pattern shuffle reductions; the column remainder (10) is handled by a shortened final iteration.

2 Results and Data Analysis

This section reports experimental results and analyzes observed latency and jitter. All timings are measured on an NVIDIA Tesla T4 (compute capability 7.5). Latencies are reported in microseconds (μs) with percentiles p50/p90/p99 over many iterations.

2.1 Experimental procedure

Each benchmark warms up the GPU before timing. Kernel-only measurements use CUDA events around the GPU work. End-to-end measurements include host–device transfers (H2D and D2H) and synchronization, reflecting practical tick-by-tick latency. Correctness is validated by comparing outputs to a float32 CPU reference implementation using the maximum absolute error.

2.2 Kernel launch overhead

Before comparing inference implementations, we quantify the baseline cost of launching a GPU kernel. Table 1 summarizes the measured latency of an empty kernel launch, both as single launches and amortized across a loop of 32 launches inside the timed region.

Table 1: Empty kernel launch latency on Tesla T4. The inner_reps=32 measurement amortizes timing overhead and provides a more stable estimate of launch cost.

Configuration	p50	p90	p99	min	max	mean	sd
inner_reps=1	4.864	5.760	6.336	4.064	65.600	4.981	0.938
inner_reps=32	2.433	2.641	3.549	2.315	24.175	2.518	0.329

Interpretation. For batch-1 pipelines consisting of multiple small kernels, a few microseconds of launch overhead per kernel can be a dominant fraction of total latency. This motivates fused kernels and CUDA Graph replay.

2.3 Kernel-only inference latency (device-resident inputs)

Table 2 compares kernel-only latency for three approaches: cuBLAS pipeline, custom warp-per-output, and two versions of the MByV-style thread-per-output approach (original and packed-weight).

Table 2: Kernel-only latency for a 3-layer MLP ($512 \to 256 \to 64 \to 1$). "Direct" uses standard launches; "CUDA Graph" replays a captured sequence.

Method	Mode	p50	p90	p99	min	max	mean	sd
cuBLAS baseline	Direct	24.384	33.856	43.008	20.640	155.424	26.505	5.359
cuBLAS baseline	CUDA Graph	14.288	14.544	17.040	12.496	307.248	14.321	2.645
Custom MByV fused (staging weights)	Direct	76.784	78.416	82.000	74.448	537.984	77.211	4.005
Custom MByV packed weights	Direct	20.480	33.088	49.152	18.912	63.904	24.548	7.306
Custom MByV packed weights	CUDA Graph	18.848	19.872	20.576	18.432	108.512	19.070	1.149
Custom warp-per-output	Direct	13.952	20.480	25.056	10.240	2050.300	16.506	20.611
Custom warp-per-output	CUDA Graph	8.448	9.440	10.240	8.192	367.872	8.688	2.716

What changed and why it worked. The original MByV implementation re-staged weights into a scratch structure each inference and read them back during compute, adding extra memory traffic. Additionally, with row-major weights, the thread-per-output mapping caused non-coalesced weight loads across lanes. Packing weights into a K-major, M-padded layout keeps the same thread mapping but converts the weight reads into coalesced accesses, reducing latency by more than $5\times$ (106.784 µs $\rightarrow$ 20.480 µs p50). CUDA Graph replay further reduces dispatch overhead for the packed MByV variant.

Best kernel-only configuration. The warp-per-output kernel with CUDA Graph achieves the best kernel-only latency (8.448 µs p50, 10.240 µs p99). The packed MByV variant is competitive with cuBLAS direct launches and demonstrates that memory layout alone can transform a batch-1 kernel.

Correctness note. The packed MByV implementation produced a maximum absolute error of $\approx 1.98 \times 10^{-3}$ versus the float32 CPU reference in this experiment. This indicates a systematic numerical or implementation difference (e.g., boundary handling with padding, or differences in floating-point operation ordering). In low-latency financial signal settings, such small deviations can be acceptable depending on model sensitivity, but they should be investigated and bounded when adopting mixed layouts or padding.

2.4 End-to-end latency (including H2D/D2H)

Table 3 reports end-to-end latency for the custom warp-per-output MLP, including host–device transfers.

Table 3: End-to-end latency for the custom warp-per-output MLP, including H2D and D2H. Pageable vs. pinned host memory and CUDA Graph replay are compared.

Method	Host/Launch mode	p50	p90	p99	min	max	mean	sd
Custom warp-per-output	Direct + pageable	25.920	34.560	43.520	18.080	420.736	28.325	6.382
Custom warp-per-output	Direct + pinned	27.840	28.224	35.040	14.144	133.824	25.344	4.451
Custom warp-per-output	CUDA Graph + pinned	21.632	27.904	28.288	12.384	476.960	21.971	4.452

Interpretation. Pinned host memory improves tail latency (p99) compared to pageable memory by reducing internal staging and enabling more predictable DMA behavior. Capturing transfers and kernels into a CUDA Graph yields the best end-to-end median and improves p99.

2.5 Persistent kernel with zero-copy mapped memory

Table 4 evaluates an optional persistent-kernel streaming design, where a single long-lived kernel polls a mapped host buffer for new ticks.

Discussion. This design eliminates per-tick kernel launches, but reading the input vector across PCIe (zero-copy) is slower than copying to device memory for this workload. The method is nevertheless valuable as an architectural demonstration of persistent execution and can be combined with device-resident staging buffers in a hybrid design.

Table 4: Persistent-kernel streaming inference using zero-copy mapped host memory (one kernel running continuously).

p50	p90	p99	min	max	mean	sd
52.659	57.119	115.533	48.895	1866.059	58.598	38.295

3 Conclusion

The purpose of this assignment was to build and evaluate a GPU-accelerated inference engine optimized for ultra-low-latency market-signal prediction at batch size 1, and to understand which overheads dominate microsecond-scale inference.

A custom warp-per-output dense-layer kernel with fused bias and activation outperformed a cuBLAS-based baseline, and CUDA Graph replay substantially reduced overhead for both implementations.

A key additional lesson came from improving the MByV-style thread-per-output kernel without changing its thread mapping: by changing only the weight storage format to a packed K-major layout, kernel-only latency improved significantly, demonstrating that memory coalescing can dominate performance for batch-1 inference.

Overall, the main conclusion is that batch-1 inference is often limited by dispatch overhead and memory movement rather than raw compute throughput. Kernel fusion and careful memory layout are effective tools for achieving stable microsecond-scale latency suitable for real-time market-signal pipelines.