# Scalable Convolutional Neural Network Inference for High Performance Computing

Justus Grabowsky & Badie Taye

September 14, 2025

**Abstract**

This work investigates scalable inference for convolutional neural networks (CNNs) on ETH Zürich's Euler high performance computing cluster. Both single and multi-threaded CPUs and GPUs are targeted, with a focus on identifying and exploiting computational patterns that can be optimized for efficiency. A minimal CNN architecture was implemented across seven variants, covering naive and stencil CPU models, OpenMP parallel CPU models, and a CUDA GPU model. Benchmarks using the MNIST dataset show that the stencil formulation improves cache locality and single-core performance compared to the naive one. OpenMP threading provides sublinear speedups that plateau once memory bandwidth is saturated. The CUDA implementation shows the highest throughput once scaled across multiple devices, leveraging greater memory bandwidth and fine-grained parallelism. These results highlight the bandwidth-limited nature of CNN inference on CPUs and the scalability advantages of GPUs. Extending to more complex architectures and multi-node settings will provide a fuller picture of performance for large-scale scientific computing.

## Contents

# 1 Introduction

Convolutional neural networks (CNNs) have gained increasing traction in scientific computing due to their ability to extract hierarchical spatial features from structured data. For instance, CNNs are being integrated into weather and climate modeling workflows to accelerate components such as the parameterization of subgrid processes, emulation of physical models, and data assimilation. Their inherent locality and translational invariance make CNNs well suited for representing physical fields such as temperature, pressure, or precipitation.

Because these models are deployed in large-scale simulations on high-performance systems, the efficiency of inference becomes critical for scalability and resource utilization. Focusing on performance is therefore not merely a matter of speed but a necessity for enabling real-time analysis, large-scale surrogate models, and the integration of machine learning components into production scientific workflows.

This project investigates the development and performance optimization of CNN forward inference in C++. Our motivation for implementing CNNs in C++ is that it enables direct control over memory management and parallelization strategies, which is essential for exploiting modern CPU and GPU architectures. The primary objective is to evaluate how various implementations of CNN inference perform across CPU and GPU architectures, with a focus on identifying and exploiting computational patterns that can be optimized for scalability and efficiency. Using the MNIST dataset as a representative benchmark, we construct a minimal yet functional CNN architecture to simulate forward inference.

The CPU implementation serves as a baseline and progresses through a sequence of increasingly optimized variants. Starting with a naive serial implementation, we introduce a stencil-based version to better reflect the regular memory access patterns of convolution operations. These implementations are then further enhanced using OpenMP parallelism, enabling shared-memory multicore scalability. These CPU versions provide important insights into performance bottlenecks associated with memory access and loop structures in convolutional workloads.

Building upon the CPU implementation, the project transitions to GPU-based implementations using CUDA, where the focus shifts toward maximizing memory throughput and exploiting thread-level parallelism. The convolution kernel is redesigned to fit the GPU memory model, with optimizations such as tiling, shared memory usage, and coalesced memory access. Together, these implementations establish a controlled progression from simple baselines to optimized CPU and GPU kernels, providing a clear framework for analyzing performance and scalability across both architectures.

# 2 Methodology

## 2.1 CNN Architecture & Input Data

The CNN architecture is a simple two-block design intended as a baseline for inference benchmarking. It comprises two convolutional layers, each followed by ReLU and 2×2 max-pooling, and a final fully connected output layer. The first convolution expands the input from one channel to eight feature maps with 3×3 kernels. A ReLU introduces nonlinearity, and 2×2 pooling halves the spatial resolution. The second convolution increases the feature maps from eight to sixteen (also with 3×3 kernels), followed by ReLU and 2×2 pooling to further downsample the representation. The resulting feature maps are flattened into a single vector and passed through a fully connected layer that produces ten output values (one per MNIST digit class). All convolutional and fully connected layers include weights and biases. These parameters remain fixed throughout execution, with no training or updates. This ensures consistency and reproducibility in benchmarking. Normalization and dropout layers are omitted to keep the architecture minimal and interpretable for performance evaluation. We use the MNIST handwritten digits dataset, a widely adopted benchmark of 70,000 grayscale images of size $28 \times 28$ across ten classes (digits 0–9) [1]. In our experiments, MNIST is used strictly for inference with randomly fixed weights.

## 2.2 Implementation Tracks

We organize the seven implementations into four tracks:

- CPU – simple baselines (naive, stencil): Two serial versions used as a baseline and reference timings. The naive baseline expresses convolution and dense layers as direct nested loops. The stencil baseline reorganizes the convolution into a sliding-window (stencil) form to emphasize spatial/temporal locality and more regular memory access.
- CPU – parallel (naive + stencil): The serial baselines augmented with OpenMP shared-memory parallelism. Coarse-grain parallel work is exposed over batch items and output channels; fine-grain vectorization is encouraged on unit-stride inner loops. These two versions isolate the effect of threading without additional optimizations.
- CPU – parallel + optimization (naive + stencil): Builds on the parallel versions while preserving the same parallel domains. The goal is to lower constant factors and improve robustness without changing algorithms or data layouts.
- CUDA GPU: This version leverages CUDA to parallelize computations across GPU threads. It incorporates techniques such as thread block tiling, shared memory usage, and memory coalescing to fully exploit the GPU's high-throughput architecture.

## 2.3 Design Rationale and Performance Expectations

The implementation is guided by performance expectations based on known architectural characteristics of CPUs and GPUs. On CPUs, the design moves from a naive baseline to a stencil formulation to improve cache behavior: the naive loops incur non-contiguous accesses and repeatedly reload overlapping windows, whereas the stencil reorders loops to keep the width dimension unit-stride and reuse overlapping pixels while they remain in cache. This is expected to reduce miss rates and lower single-core compute time.

Parallelization exposes independent work across batch items and output channels. We expect sublinear speedups that saturate as memory bandwidth is approached. Since the stencil version already improves locality on one core, it reaches the bandwidth ceiling earlier and thus shows smaller relative speedups. The naive version, being less efficient initially, leaves more room for threads and appears to scale better, although starting from a weaker baseline. Finally, micro-optimizations target constant factors. We expect small improvements in absolute time but no change in the bandwidth-limited scaling regime.

On the GPU, the implementation is structured to exploit parallelism, with each thread responsible for computing a single output element. Performance is expected to benefit significantly from coalesced global memory access, reduced memory latency through shared memory tiling, and overall high memory throughput.

# 3 Implementation & Optimizations

## 3.1 Initial CPU Implementation

The starting point was a naive implementation of the CNN. In this baseline, all layers were realized as explicit nested loops that directly compute each output element. For the convolution layers, this involves iterating over all spatial positions, channels, and kernel elements, and explicitly summing the products of input pixels and filter weights. The corresponding code structure is shown in Figure 1a. The design is straightforward and mirrors the mathematical definition of a convolution, serving as a clear reference and baseline for later implementations. However, it is also inefficient: each output value requires multiple redundant memory accesses, neighboring convolution windows repeatedly reload overlapping input regions, and memory is accessed in a non-contiguous manner, which results in poor cache utilization. The naive implementation therefore ensures correctness but offers limited performance.

To address these inefficiencies, a stencil-based reformulation of the convolution operation was introduced. The stencil abstraction originates in scientific computing, where similar sliding-window patterns are used in finite-difference discretizations of partial differential equations. By reorganizing the loops to follow this pattern, the computation emphasizes both spatial and temporal locality, increasing the likelihood that recently accessed data remain in cache for subsequent operations. The resulting code fragment, illustrated in Figure 1b, explicitly unrolls the $3 \times 3$ kernel accesses to reduce redundant indexing and better exploit memory locality. The stencil formulation does not alter the computation's results but makes the structure more consistent with high-performance kernels.

```
for (int oc = 0; oc < out_channels_; ++oc) {
    for (int i = 0; i < H_out; ++i) {
        for (int j = 0; j < W_out; ++j) {
            float sum = biases_[oc];
            for (int ic = 0; ic < in_channels_; ++ic) {
                for (int m = 0; m < kernel_size_; ++m) {
                    for (int n = 0; n < kernel_size_; ++n) {
                        const int row = i * stride_ + m;
                        const int col = j * stride_ + n;
                        sum += in[ic][row][col] * weights_[oc][ic][m][n];
                    }
                }
            }
            out[oc][i][j] = sum;
        }
    }
}
```

**Listing (1)** Naive convolution implementation

```
for (int oc = 0; oc < out_channels_; ++oc) {
    for (int i = 1; i < H_out + 1; ++i) {
        for (int j = 1; j < W_out + 1; ++j) {
            float sum = weights_[oc][0][0][0] * in[0][i - 1][j - 1] +
                        weights_[oc][0][0][1] * in[0][i - 1][j] +
                        /* ..... */
                        weights_[oc][0][2][1] * in[0][i + 1][j] +
                        weights_[oc][0][2][2] * in[0][i + 1][j + 1] + biases_[oc];
            out[oc][i - 1][j - 1] = sum;
        }
    }
}
```

**Listing (2)** Stencil convolution implementation

**Figure 1:** Comparison of convolution implementations. (a) Naive version with nested loops over all input channels and kernel elements. (b) Stencil version with an explicitly unrolled $3 \times 3$ access pattern (middle taps omitted for brevity).

## 3.2 Parallelized CPU Models

BBuilding on the sequential versions, shared-memory parallelism was introduced using OpenMP. Both the naive and stencil implementations were parallelized, enabling evaluation of parallelism on two different baseline structures. The naive version benefited mainly from distributing work across batch elements and output channels, while the stencil version combined the same coarse-grain strategy with more cache-friendly loop ordering.

For convolution layers, the most natural parallelization domain is the batch dimension, since each image can be processed independently without risk of race conditions. Parallelism over output channels was also introduced, further distributing work across threads. At a finer granularity, inner width loops were vectorized using `#pragma omp simd`.

The activation functions and pooling layers were parallelized using a similar principle. In ReLU, outer loops over rows or feature maps were distributed across threads, while the innermost loops were vectorized to exploit data-level parallelism. For max pooling, per-image work was assigned to separate threads, whereas the small $2 \times 2$ pooling window was handled with SIMD rather than threading, since its size is too small to justify explicit parallelization. This design avoids oversubscription while still capturing available instruction-level parallelism.

The fully connected layer was parallelized by collapsing loops over batch samples and output neurons, ensuring that each thread computes independent outputs. The innermost accumulation loop was expressed with a SIMD reduction, which maps well onto the unit-stride memory layout of the input features. This exposes ample parallel work even for modest batch sizes, while ensuring efficient use of vector registers.

Across both variants, we maintained clarity and portability. The parallel regions used `default(none)` with explicit shared variable declarations, and static scheduling was applied consistently since the workload is highly regular. Small problem sizes were guarded against parallel overhead by conditional `if(...)` clauses, and nested parallel regions were explicitly avoided to prevent oversubscription.

## 3.3 Refined CPU Optimizations

The fully optimized CPU versions retained the same parallelization strategy as before but applied a series of smaller refinements to reduce runtime overhead. Outputs were pre-allocated before entering parallel regions, loop invariants were hoisted, and row references were cached to reduce redundant indexing. Selective `#pragma omp simd` hints were added to unit-stride inner loops such as dot products and copies, while branchless clamps simplified ReLU. In convolution, padding was prepared once per sample and indexing was streamlined. In pooling, the small window used SIMD rather than threading. Headers were cleaned and heavy includes were moved to source files. These changes did not alter the algorithms or parallel domains, but made the code more efficient and scalable in practice.

## 3.4 CUDA GPU

The transition from a CPU-based to a GPU-based implementation in the project primarily aimed to exploit the inherent parallelism and high-throughput capabilities of modern GPUs to accelerate the computational workload of a convolutional neural network (CNN). So far, the entire inference process comprising convolution, activation, pooling, and fully connected layers was implemented in C++ for execution on the CPU. While this approach is straightforward and benefits from well-optimized serial execution (as discussed earlier), it inherently suffers from limited concurrency and memory bandwidth, particularly when processing datasets such as MNIST. The serial nature of the CPU implementation becomes a bottleneck, especially under batch workloads or when scaling up the problem size, and this necessitates a shift to GPU acceleration.

To achieve this, key operations were reimplemented using CUDA C/C++ in separate `.cu` files. Implementation wise, the CPU kernels are replaced with CUDA kernels and launch sites. Compute stages are expressed as `__global__` functions and invoked with `<<<grid, block>>>`, with thread/block indexing replacing the CPU's nested loops over `(n, c, y, x)`. Host code switches from stack/heap allocations to `cudaMalloc` / `cudaFree` for device buffers and from direct pointer arithmetic to explicit `cudaMemcpy` (or its

async variant) at batch boundaries; correctness is guarded with standard CUDA error checks and a synchronization point (e.g., `cudaDeviceSynchronize()`) before copying results to host. Indexing math and tensor layout are aligned so that consecutive threads access consecutive elements (coalesced reads/writes), and constants that do not change across `CHUNK` iterations (e.g., layer weights and biases) are kept device-resident to amortize transfers over the `TOTAL` / `CHUNK` loop.

The GPU path rewrites the inference loop so that model parameters are copied once to device memory and reused across the entire run, while inputs are streamed in batches controlled by the binary's two arguments: `TOTAL` (samples to evaluate) and `CHUNK` (batch size per device iteration). For each batch, the GPU version allocates (or reuses) device buffers for activations, performs the full forward pass on device, and copies only the logits back to host. This removes per-layer round-trips present in, for instance, CPU versions and makes `CHUNK` the primary lever for trading memory footprint against throughput.

A few limitations are present in the current approach. The reliance on `SLURM_PROCID` assumes that the job is launched with multiple ranks, but the benchmarking script sets `SLURM_PROCID=0` unconditionally, so multi-GPU scaling requires further orchestration inside the binary. Moreover, no explicit device visibility (`CUDA_VISIBLE_DEVICES`) is enforced, so binding depends on system defaults. Finally, while `cudaSetDevice` enables multi-device execution, inter-device communication is absent; scaling is therefore limited to data parallelism where each GPU processes an independent chunk of samples.

# 4 Numerical Experiments & Performance Benchmarking

## 4.1 Euler Cluster xPU Architectures

We executed our benchmarks on ETH Zürich's Euler cluster [2], a centrally managed HPC system. Euler is a heterogeneous cluster with distinct CPU and GPU partitions accessible through the Slurm scheduler. The system provides shared software stacks and a standard batch/interactive workflow for CPU and GPU jobs.

Euler's CPU nodes are dual-socket AMD EPYC 7742 ("Rome") systems with $2 \times 64$ cores and DDR4–3200 memory. Each socket exposes eight memory channels for a theoretical $\sim$204.8 GB/s DRAM bandwidth. With NPS = 4, the socket is split into four NUMA domains, so each domain only sees a fraction of that bandwidth. Linux uses a first-touch policy for page placement, making memory locality (threads accessing pages from their own NUMA domain) crucial.

For the multi-GPU experiments we used Euler's GPU partition with NVIDIA GeForce RTX 2080 Ti devices (11 GB). These nodes are part of the documented GPU estate and provide ample device memory and FP32 throughput.

## 4.2 CPU Benchmarking

We benchmarked six CPU binaries that cover two serial baselines (naive and stencil) and four parallel models (naive with simple parallelization, naive parallelized and optimized, stencil with simple parallelization, stencil parallelized and optimized). Serial models were fixed to one thread, while thread sweeps were applied to the parallel models. Threads were pinned to CPU cores and internal threading of math libraries was disabled so that OpenMP was the only source of parallelism.

Table 1 and Figures 2 and 3 summarize the results. Figure 2 shows compute time versus thread count with interquartile range (IQR). Compute time isolates the kernels and is used for scaling analysis. Table 1 reports the best total wall time, thread counts achieving them, as well as the corresponding speedup and efficiency computed relative to the serial baseline (efficiency = speedup / threads). Total time reflects end-to-end throughput including I/O, memory allocation, and other fixed costs. Figure 3a shows speedup computed from the compute time medians with respect to one thread and Figure 3b shows efficiency. Each point is the median over 10 runs with 50 000 samples following a short warm-up.

In Figure 2, compute time decreases with threads. The stencil curves plateau around 4 threads, while the naive curves flatten at higher thread counts. At 14 threads, compute time starts rising noticeably. This yields the best end-to-end times at 12 threads for all models (cf. Table 1). The parallelized and optimized variants provide a small improvement over the simple-parallel versions on the order of a few tenths of a second (about 3–10%), more for naive and less at higher thread counts. The IQR remains small (about 0.1 s) up to 12 threads, but grows significantly at 14 and 16 threads, with IQRs reaching up to $\sim$5 s even as median total compute times lie in the 7–11 s range.

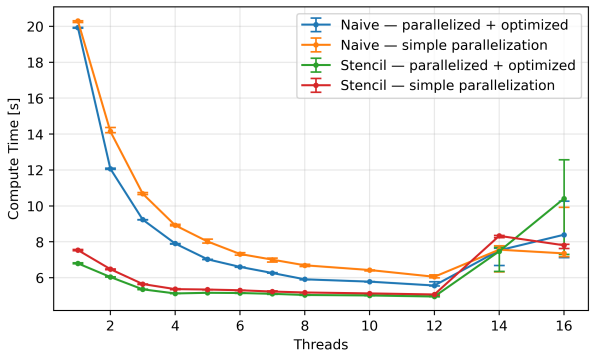
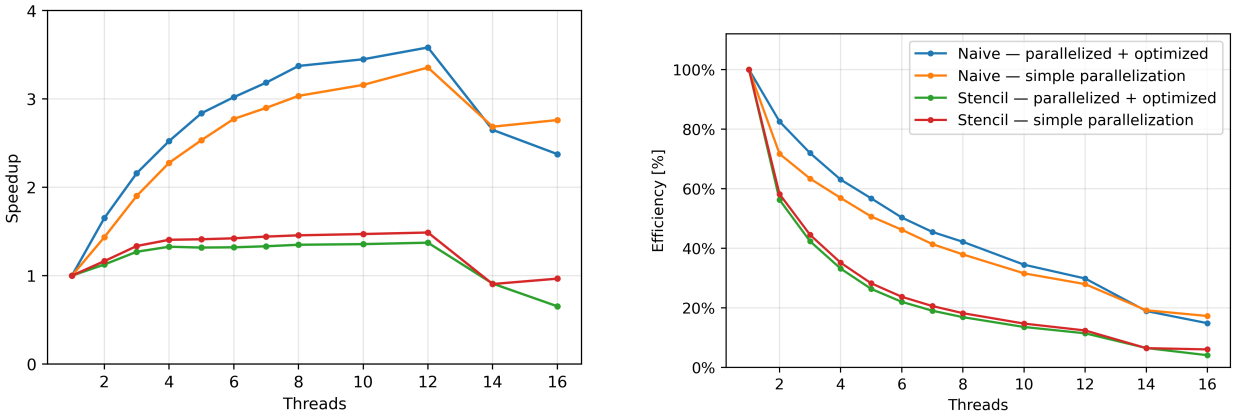



**Figure 2:** Compute time versus threads with interquartile range.

Figure 3 paints a consistent picture. The speedup from compute time medians increases with thread count but less than linearly, with the naive implementations exhibiting the stronger gains. It then plateaus and declines at the highest thread counts. Accordingly, efficiency steadily decreases as threads increase.

Table 1: Best end-to-end time per CPU model (median over repeats at the best thread count). Speedup and efficiency are computed from total wall time relative to the corresponding serial baseline; efficiency = speedup / threads.

| Model | Best time [s] | Threads | Speedup | Efficiency |
|---|---|---|---|---|
| Naive, serial | 19.945 | 1 | – | – |
| Stencil, serial | 6.364 | 1 | – | – |
| Naive, simple parallelization | 6.322 | 12 | 3.252 | 27.1 % |
| Naive, parallelized and optimized | 5.820 | 12 | 3.465 | 28.9 % |
| Stencil, simple parallelization | 5.341 | 12 | 1.359 | 12.1 % |
| Stencil, parallelized and optimized | 5.224 | 12 | 1.287 | 11.2 % |




**(a)** Speedup **(b)** Efficiency

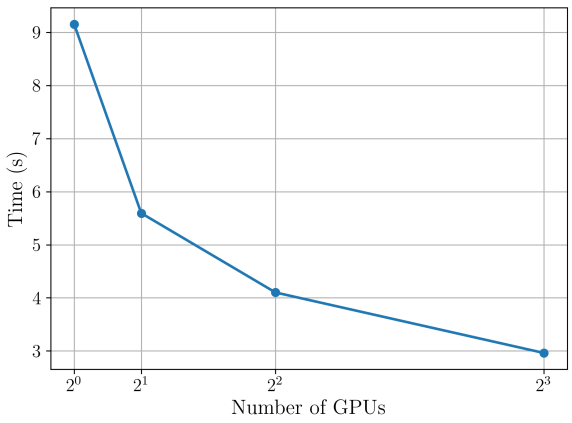
**Figure 3:** CPU scaling metrics derived from compute time medians.

## 4.3 GPU Benchmarking

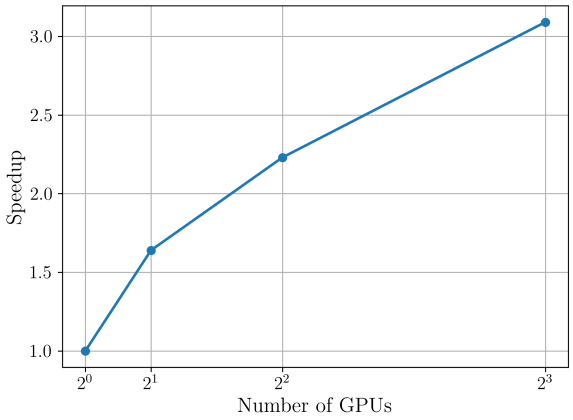
To evaluate the forward inference performance of the GPU-enabled CNN model, we conducted both strong and weak scaling experiments on the Euler Cluster. The tests were executed on NVIDIA GeForce RTX 2080 Ti GPUs, each with 11 GB of device memory, using up to 8 GPUs in parallel. The benchmarking environment was configured with CUDA version 12.8 and driver version 570.172.08, as confirmed by the command `nvidia-smi`. This setup provided a modern CUDA runtime and ensured that kernels were executed efficiently on high-memory GPUs suitable for deep learning workloads. The purpose of these experiments was to assess how well the GPU implementation scales in terms of latency and throughput under increasing device counts, and to directly contrast ideal scaling behavior with practical performance outcomes.

In the strong scaling scheme, the total dataset (50,000 samples) is fixed while the number of GPUs is increased. In the benchmarking script, this is realized by keeping `TOTAL=$BASE_SAMPLES` (50,000 samples) constant and dividing the work across devices by setting `CHUNK=$((BASE_SAMPLES / GPUS))`, capped by `MAX_CHUNK`. The aim is to observe how effectively additional GPUs shorten the execution time for the same overall workload. An ideal outcome would be linear speedup and constant efficiency, but in practice kernel launch overheads, memory transfers, and synchronization costs prevent perfect scaling. Strong scaling thus

provides a measure of how well the GPU implementation reduces latency for a fixed task size, which is directly relevant for use cases where the dataset size is not flexible.



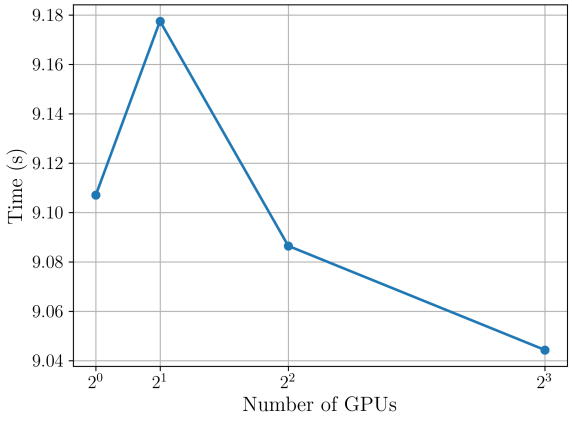

**(a)** Runtime as a function of GPU count.



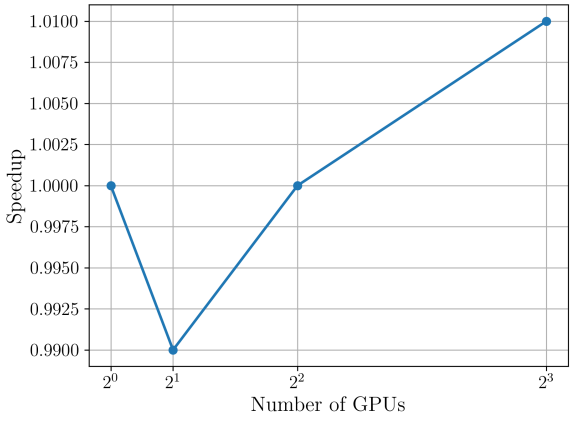

**(b)** Observed speedup relative to the single-GPU baseline.

**Figure 4:** Strong scaling performance of the CNN inference benchmark on up to 8 GPUs.

In the weak scaling scheme, the intent is different: the per-GPU workload is held constant while the total workload grows with the number of GPUs. Our test dataset consisted of 50,000 samples, so to apply weak scaling we logically replicated this workload by defining `CHUNK=BASE_SAMPLES` (50,000) for each GPU and setting `TOTAL=((BASE_SAMPLES * GPUS))`. In effect, a 2-GPU run processed 100,000 samples, a 4-GPU run 200,000 samples, and so on, ensuring that each GPU always handled the same amount of work as in the baseline. The benchmark then assesses whether runtime remains stable as GPUs and workload scale together. Ideally, runtimes stay flat across GPU counts, signaling that parallel execution scales without excessive coordination or transfer overhead. This makes weak scaling particularly important for evaluating throughput in large-scale deployments, where increasing data volume must be matched by proportional hardware resources.

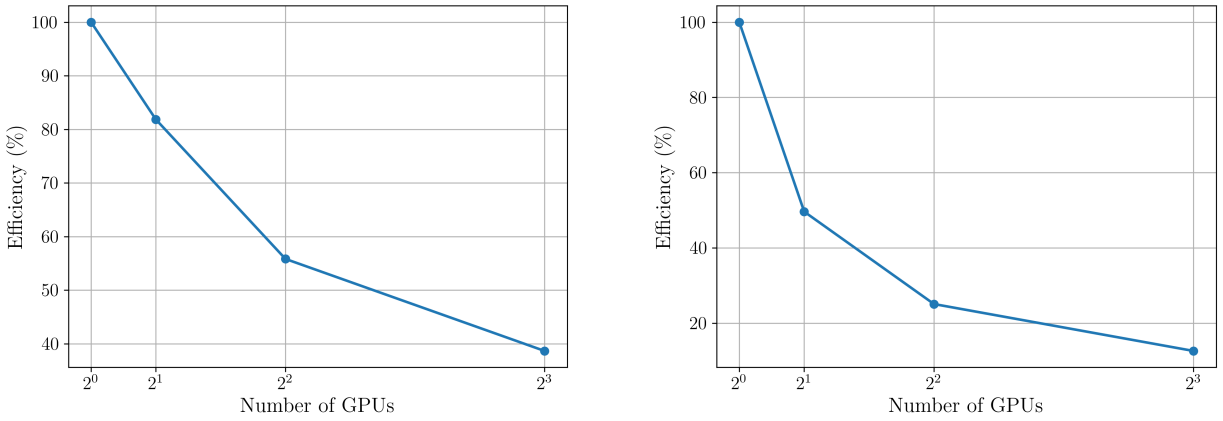



**(a)** Runtime with constant workload per GPU.




**(b)** Apparent speedup under weak scaling.

**Figure 5:** Weak scaling performance of the CNN inference benchmark as total problem size increases with GPU count.



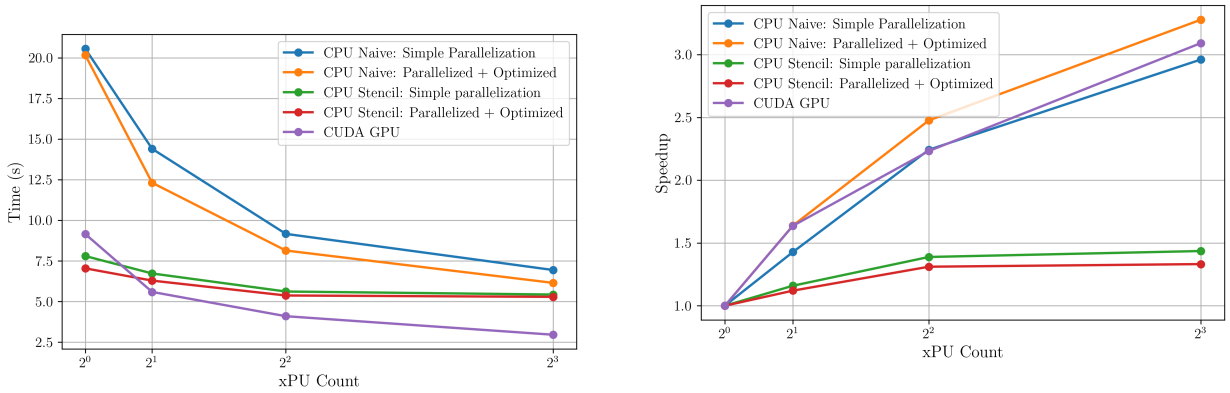

**(a)** Parallel efficiency under strong scaling.

**(b)** Parallel efficiency under weak scaling.

**Figure 6:** Efficiency of the CNN inference benchmark under strong and weak scaling on up to 8 GPUs.

## 4.4 CPU-GPU Benchmark Comparison

To better visualize how the CPU and GPU implementations compare alongside each other, Figure 7 shows runtime and speedup as the number of CPUs and GPUs is increased in the same plot. At one unit, the naive CPU baseline is slowest, followed by the GPU, while the stencil CPU variants achieve the best single-unit times. As the number of units grows, the GPU shows the strongest scaling and becomes the fastest overall, while the CPU curves improve but eventually level off at higher thread counts.




**(a)** Runtime as a function of xPU count.

**(b)** Observed speedup relative to the single-xPU baseline.

**Figure 7:** Scaling performance of the CNN inference benchmark as xPU count increases fixed dataset size.

# 5 Discussion

## 5.1 CPU Benchmarking

The stencil formulation is faster than the naive baseline because it improves spatial and temporal locality. The naive loops repeatedly reload overlapping input windows and access memory in a non-contiguous pattern, which causes poor cache reuse. The stencil version restructures the access so that the width dimension is unit-stride, overlapping pixels are reused while they are still in cache, and index arithmetic inside the hot loops is reduced. This lowers single-core compute time and explains the gap between the serial models in Figure 2 and Table 1.

The scaling plots in Figure 3 show a clear contrast between the naive and stencil variants. This difference reflects how close each kernel is to saturating memory bandwidth on a single core. The stencil version already approaches the bandwidth roof on a single core because of better locality, so additional threads mostly compete for the same memory bandwidth and scaling saturates early. The naive version is less efficient on one core, leaving headroom when threads are added. Coarse-grain parallelism over batch items and output channels exposes independent work with disjoint writes, and fine-grain vectorization along the unit-stride width dimension helps within each thread. As threads increase, speedup rises but remains below perfect linear scaling, and efficiency falls, which is consistent with a memory-bound workload with small fixed overhead outside the kernels.

Even with threading controls, variability increases noticeably at 14–16 threads. This is likely due to how memory is organized on Euler's dual-socket CPUs. Each socket is divided into NUMA domains, and once threads span multiple domains, access times depend on where data pages were first placed ("first-touch"). Small differences in how threads are scheduled can therefore lead to different fractions of remote memory traffic and greater spread in runtimes. In addition, CPU affinity can vary slightly across runs, so threads may end up sharing caches or memory controllers differently, further amplifying variance. Finally, nodes are not necessarily exclusive. Co-tenant jobs can contend for shared resources and widen the IQRs observed at high thread counts on bandwidth-bound kernels.

The refined CPU versions keep the same parallel domains and focus on constant factors. Pre-sizing outputs, hoisting loop invariants, using branchless clamps in ReLU, adding selective SIMD on unit-stride reductions, and avoiding allocations in hot paths lower the single-core time and reduce per-thread overhead. In the plots, this appears as a slight downward shift of the compute time curves with a similar slope, while the best total times improve by modest margins. This slight improvement is expected for bandwidth-bound kernels. As a result, the speedup improves slightly over the simple parallel version, while the overall scaling slope remains similar.

## 5.2 GPU Benchmarking

The strong scaling results show that performance improves consistently as more GPUs are employed, though the gains are sublinear. Starting from a baseline of 9.16 s on a single GPU, the runtime decreases to 5.59 s with two GPUs, giving a speedup of 1.64× and an efficiency of about 81.8%. As the model is scaled further, the runtime continues to drop to 4.10 s on four GPUs and 2.96 s on eight GPUs. Although the efficiencies decrease (55.8% and 38.6% respectively), the monotonic runtime improvement demonstrates that the implementation scales in the intended direction.

The reduced efficiency at higher GPU counts reflects the challenge of strong scaling with a fixed workload. As the number of devices increases, each GPU is assigned progressively smaller chunks of work, and overheads such as memory transfers, kernel launches, and device synchronization grow in relative importance. This explains why the speedup slows down with increasing GPUs rather than approaching the ideal of 8×. In addition, the NVIDIA GeForce RTX 2080 Ti GPUs on Euler, while providing 11 GB of memory per device, do not provide high-bandwidth interconnects such as NVLink. Communication between GPUs must therefore pass through the PCIe bus, introducing additional latency and limiting throughput when scaling to multiple devices. These hardware characteristics, combined with the fixed problem size, further account for the diminishing efficiency observed at higher GPU counts. Nonetheless, the results confirm that the

GPU-enabled implementation benefits from parallelization and achieves good scaling behavior for moderate device counts, providing a reliable baseline for inference performance on the Euler cluster.

## 5.3 CPU–GPU Comparison

The workload has low arithmetic intensity, so performance is limited by memory-bandwidth rather than compute. On CPUs, once cache locality is exploited, as in the stencil kernels, a single core can already approach the bandwidth roofline, leaving little headroom for further gains. Adding threads mainly increases contention, while less efficient implementations such as the naive kernels scale further only because they start below the roofline, before converging when shared bandwidth is exhausted.

The architecture of GPUs differs fundamentally. They offer much higher device memory bandwidth, thousands of lightweight threads, and coalesced access patterns. These properties let GPUs outperform many CPU threads once overheads are amortized. Sublinear scaling arises mainly from synchronization and PCIe transfer costs, not from limits in computational parallelism. This architectural contrast explains the scaling behavior observed in our benchmarks.

# 6 Conclusions

The goal of this work was to explore how CNNs, increasingly important in scientific computing for representing structured inputs, can be implemented and optimized for high-performance systems. Efficient inference is a key requirement when CNNs are integrated into large-scale simulations, making it essential to understand how different algorithmic and architectural choices affect performance.

We developed several versions of the same small CNN: a naive baseline with deeply nested loops, a stencil reformulation designed to improve spatial and temporal locality, parallelized versions using OpenMP to leverage multicore CPUs, and finally a CUDA implementation targeting GPUs. The optimized CPU variants retained the same algorithmic structure but applied low-level refinements. On the other hand, the GPU implementation replaced serial C++ kernels with CUDA implementations that offload convolution, activation, pooling, and fully connected operations to device memory, using thread/block parallelism, coalesced memory access, and batched execution to exploit GPU throughput and host-device transfers.

On CPUs, the stencil formulation delivers the best single-core time by improving spatial and temporal locality, while the naive baseline starts much slower. With threading, both families see sublinear speedup and eventually saturate almost at the same memory bandwidth roofline. Best end-to-end times occur using 12 threads. On a single unit, the CPU stencil achieves the fastest time by exploiting cache locality, but from two or more GPUs onward the GPU implementation is consistently the best, as fixed overheads are amortized and its higher bandwidth and parallelism dominate. GPU efficiency declines with device count due to coordination and PCIe transfers, yet throughput remains about twice that of the fastest CPU case at the largest scale tested. Overall, the trends reflect a bandwidth-bound workload where cache-aware kernels cap CPU gains early, while GPU memory bandwidth and concurrency dominate once the batch is large enough.

A natural next step is to move beyond this minimal CNN toward deeper architectures and more realistic scientific workloads. Future work should explore larger models and datasets that stress both compute and memory systems, and extend the implementation beyond inference to include training. On the implementation side, techniques such as asynchronous transfers, stream-based kernel execution, and refined batching could reduce overheads and improve device utilization. At the system level, scaling beyond a single socket or node through distributed data parallelism will be essential, together with communication libraries that overcome the limitations of PCIe-based interconnects. Finally, strengthening NUMA control on CPUs, and comparing against optimized libraries would give a more comprehensive picture of performance in practical deep learning workflows.

# References

[1] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[2] ETH Zürich Scientific IT Services. Euler: Hpc documentation. https://docs.hpc.ethz.ch/, 2025. Accessed: 2025-09-14.