Project 3 final

Charles

March 2025

1 Introduction

I use cuBLAS to accelerate the computations by efficiently handling matrix multiplications (`cblas_sgemm`), vector additions (`cblas_saxpy`), and scaling (`cblas_sscal`) on the GPU. This allows for optimal parallel processing while reducing memory bottlenecks. By fusing matrix multiplications with bias addition and activation functions like ReLU and Softmax, I minimize redundant memory transfers and improve efficiency.

In the forward pass, I execute dense layers efficiently by parallelizing matrix operations and applying activations with vectorized functions. During backpropagation, I speed up gradient calculations using cuBLAS's optimized transposition and dot products. I also use OpenMP to parallelize CPU-side preprocessing so that data is readily available when needed by the GPU.

For GPU acceleration, I use `nvcc` with flags such as `-O3` for aggressive optimization, `--use_fast_math` to enable approximate mathematical functions, and `--extra-device-vectorization` to exploit additional vectorization. The `-Xptxas=-dlcm=ca` flag optimizes memory caching, while `--ptxas-options=-v` gives output for debugging register usage. I also enable `--fmad=true` to fuse multiply-add operations for improved throughput and apply `-gencode` flags to target specific CUDA architectures.

On the CPU side, I compile with `icpx` (Intel oneAPI's C++ compiler) and apply flags like `-O3` for high optimization levels, `-fopenmp` for multi-threading support, and `-ftree-vectorize` to enable automatic loop vectorization. Additionally, `-ffast-math` allows the compiler to reorder floating-point operations for better performance, while `-funroll-loops` reduces loop overhead by expanding iterations. The `-march=native` flag ensures the compiler generates instructions optimized for the host CPU, maximizing computational efficiency.

One of the key mistakes I made was **allocating memory multiple times in batches**, which led to fragmentation and unnecessary overhead instead of preallocating memory efficiently. Additionally, my **initial data representation choices were suboptimal**, as I didn't use **flat 2D arrays** from the start, leading to inefficient memory access patterns and unnecessary complexity in indexing. Another issue was **not setting CPU affinity**, causing threads to share the same core instead of being properly distributed across available cores,

leading to poor parallel performance. I also made the mistake of **using different BLAS libraries for different operations**, which likely introduced inconsistencies in performance and numerical precision. Furthermore, **excessive data transfers** between the CPU and GPU severely bottlenecked performance, highlighting the need for more efficient memory management and kernel execution strategies. Lastly, my **code was not cache-aware**, resulting in frequent **cache misses** due to poor data locality, significantly slowing down execution when working with large datasets.

2 Performance

Version	**Processor**	**Accuracy**	**Grind rate**	**Training Time**	**TPB or CPU cores**
GPU native	rtx6000	96.05	$1.65 * 10^6$	1.81s	256
GPU cuBLAS	rtx6000	96.17	$2.267 * 10^6$	1.32s	-
CPU native	caslake	96.07	51357.1	58.42s	16
CPU BLAS	caslake	96.14	183964	16.30s	16

Table 1: Comparison of different training versions. For cuBlas I'm relying on the library and for native I am calculating the grid and the block. I used the openblas module

3 Validation Loss

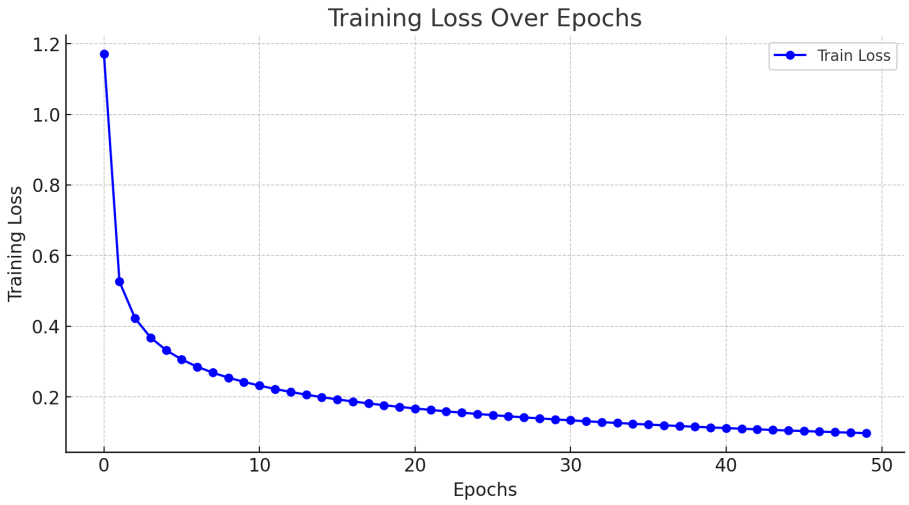

Figure 1: Training Loss

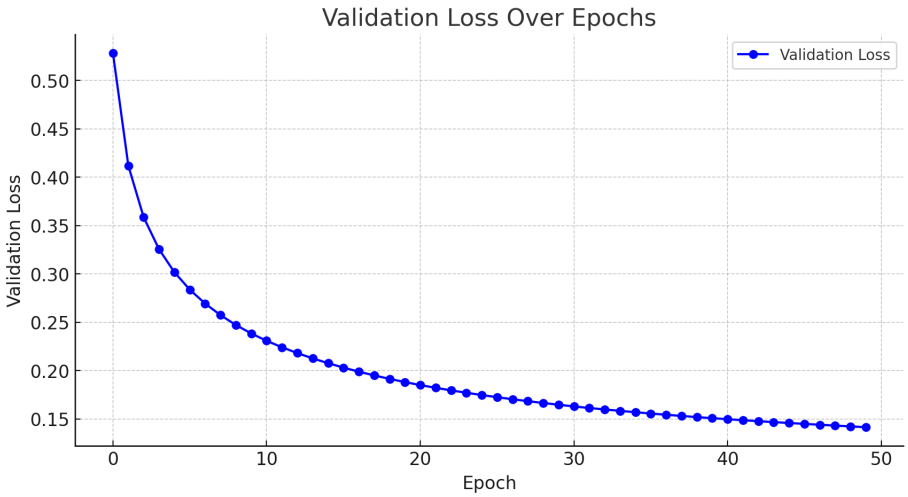

Figure 2: CPU Native

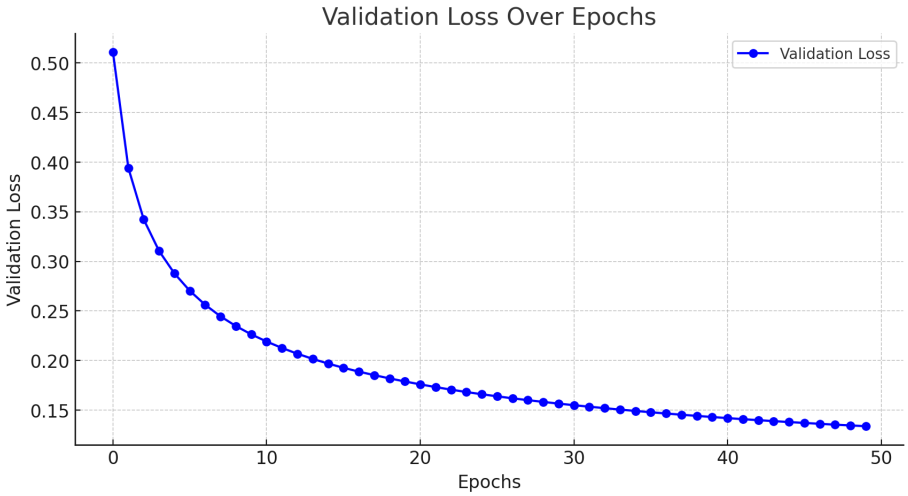

Figure 3: CPU Blas

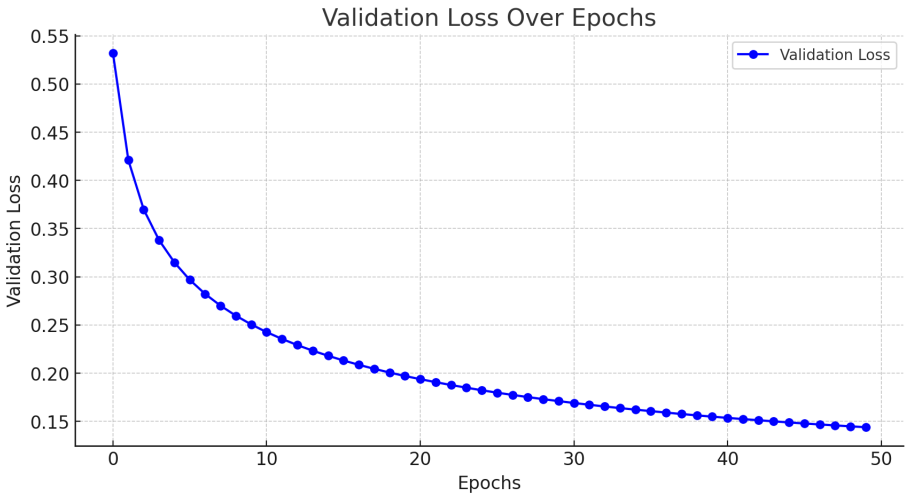

Figure 4: GPU Native

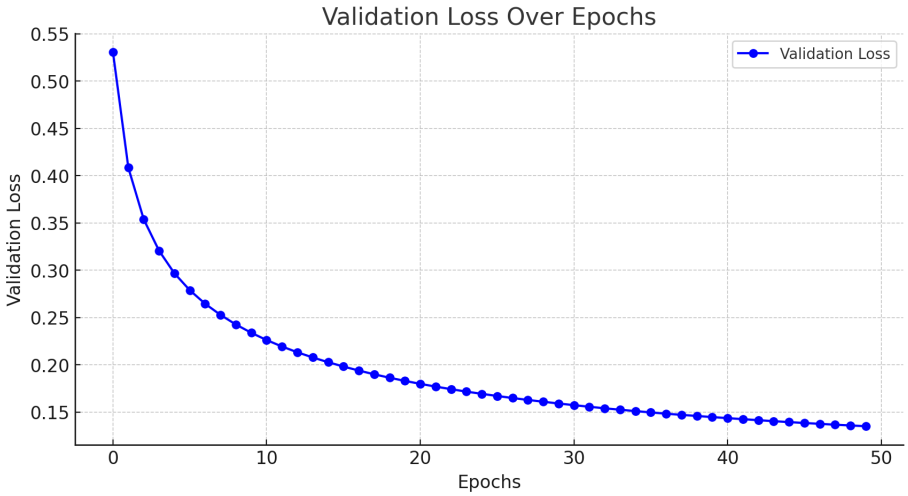

Figure 5: GPU Blas