Mobile Computing, Laboratory

Sparse Matrices

David Mihola
12211951

June 14th, 2023

Contents

1 Introduction

The main purpose of this project was to research different memory layouts and algorithms for sparse matrices. Our second goal was to implement the promising algorithms we discovered on an Android phone. Lastly, we aimed to compare these algorithms with a well-optimized TensorFlow Lite (TFLite) library by performing inference on a fully connected neural network (NN).
To accomplish these objectives, we followed a specific approach. First, we implemented algorithms that populate the memory layout and perform matrix multiplication in C++ on an Intel CPU. Second, we conducted a brief performance analysis by performing matrix multiplication on two squared matrices, one sparse and the other dense. Third, we implemented the NN using the best-performing algorithm, incorporating fused bias addition, layer activation and multi-threading. Finally, we ported the code to an Android application to be callable from Kotlin using the Native Development Kit (NDK).

2 Memory Layouts and Algorithms

In this section, we introduce three different memory layouts along with their accompanying matrix multiplication algorithms. We begin with a layout that allows for completely random sparse matrices, where no zero values are stored. Next, we present a layout also for unstructured sparsity, but which requires a zero padding, when the non-zero values are unevenly distributed. Lastly, we describe a K in N structured sparse memory layout, which enables the use of SIMD instructions.
All of the mentioned layouts were designed to store the non-zero values (including zero padding in some cases) and necessary metadata for reconstructing the original matrix in continuous memory. Furthermore, the metadata is stored in close proximity to the data it describes, improving the memory access pattern and memory caching. All layouts primarily use the row-major storage format.

2.1 Bitmap Sparse

The in data bitmap sparse layout follows a specific structure as visualised in Figure 1. The initial portion of the continuous memory is allocated for the indices of rows, enabling semi-random access. Subsequently, each row consists of a set of 32-bit bitmaps with the non-zero values stored in between them.
The matrix multiplication algorithm implemented for this layout leverages the population count and count of leading zeros instructions. The former is used to compute the index of the stored non-zero value in the original matrix, while the later is utilized to mask the already used bits for the indexing.

2.2 Block Sparse

The structure shown in Figure 2 of the block sparse layout is determined by a parameter that specifies the number of blocks per dimension (rows for row-major and columns for column-major storage formats). The number of blocks must be selected such that all non-zero values can be stored. The

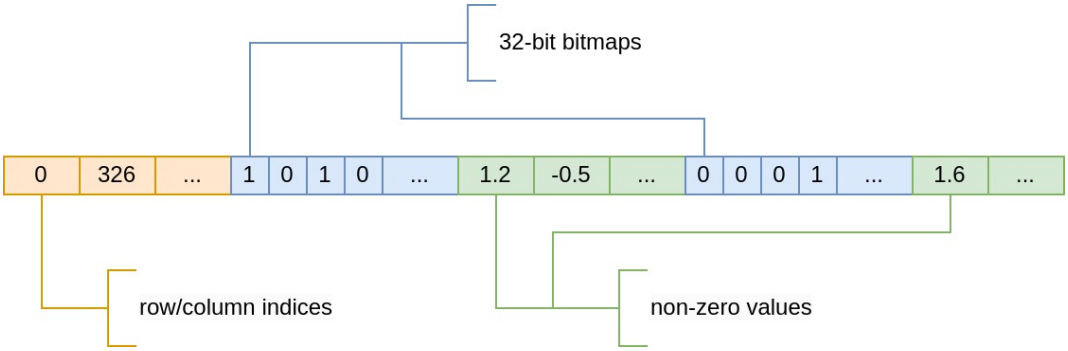

Figure 1: Bitmap sparse storage layout

limiting factors are the row/column with the largest number of non-zero values or rows/columns with unevenly distributed values. Single block can store non-zero values with offsets of up to 255 entries from its address. All blocks have the same number of entries, which are zero-padded if necessary, allowing for semi-random access. Therefore, each block stores the indices of the non-zero values within the block as well as the non-zero values itself.
The algorithm performing the matrix multiplication uses the indices of non-zero values in each block to index the multiplied matrix, while the access to the sparse matrix is continuous. Consequently, the block size is added to the address of the multiplied matrix with each processed block to ensure correct indexing.

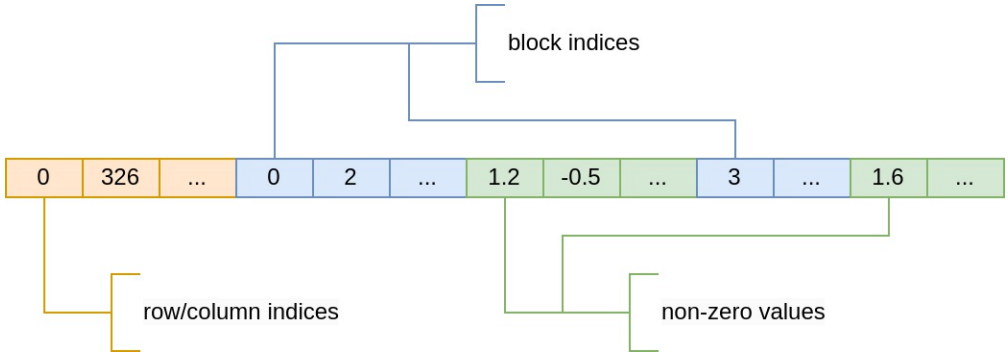

Figure 2: Block sparse storage layout

2.3 K in N Sparse

The K in N sparse pattern enforces a specific structure displayed in Figure 3, where there are always K non-zero values, zero-padded otherwise, in a block of size N (with $K \leq N$ and $N \leq 256$). For row-major storage format, the blocks span columns, while for column-major storage format, the blocks span rows. However, the entries within a block are not stored continuously in memory. The memory pattern is designed to store one value from each of the 16 blocks and 16 indices to the original matrix repeatedly for an entire row/column. This enables aligned addressing for SIMD

instructions. Consequently, the number of rows/columns of the matrix stored using this structure is reduced by a factor of N/K.
The accompanying matrix multiplication algorithm utilizes vector load instructions to read the non-zero values from both the sparse and dense matrices. It then performs element-wise multiplication using a single multiply vector instruction. The resulting vector is subsequently added element-wise to one of N accumulators based on the stored indices. This process is repeated for all these segments in a row/column and for K rows/columns of the sparse matrix, resulting in N rows/columns of the original matrix, before moving on to the next K in N block of rows/columns.

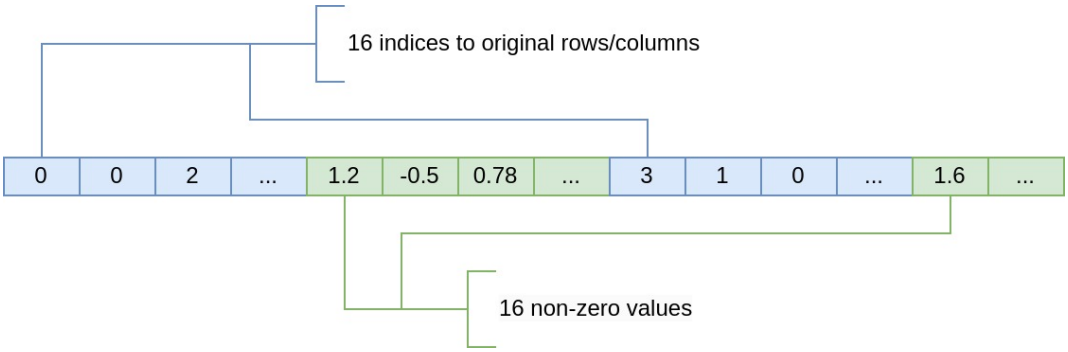

Figure 3: K in N sparse storage layout

3 Intel CPU Performance Analysis

The performance of the different storage layouts, along with their accompanying matrix multiplication algorithms, was evaluated on a square 1024 by 1024 sparse matrix dense matrix multiplication. The results of this analysis are summarized in Table 1 and depicted in Figure 4.

	0.25 sparse	**0.5 sparse**	**0.75 sparse**	**0.8125 sparse**	**0.875 sparse**
bitmap sparse	1682.46	1243.84	795.23	680.77	567.75
block sparse	728.58	517.00	309.86	253.62	208.58
K in N sparse	1282.62	636.78	204.89	129.13	81.47

Table 1: Square 1024 by 1024 sparse matrix dense matrix multiplication performance test on matrices with different sparsity percentage measured in milliseconds

Although the block sparse pattern outperforms the K in N sparse pattern on the less sparse matrices, we chose to proceed with the K in N sparse pattern for the phone implementation of the NN. It is faster for the more sparse matrices and therefore it has a better chance of beating the TFLite NN implementation. Additionally, the structured sparsity pattern allows for a better multi-threaded implementation.

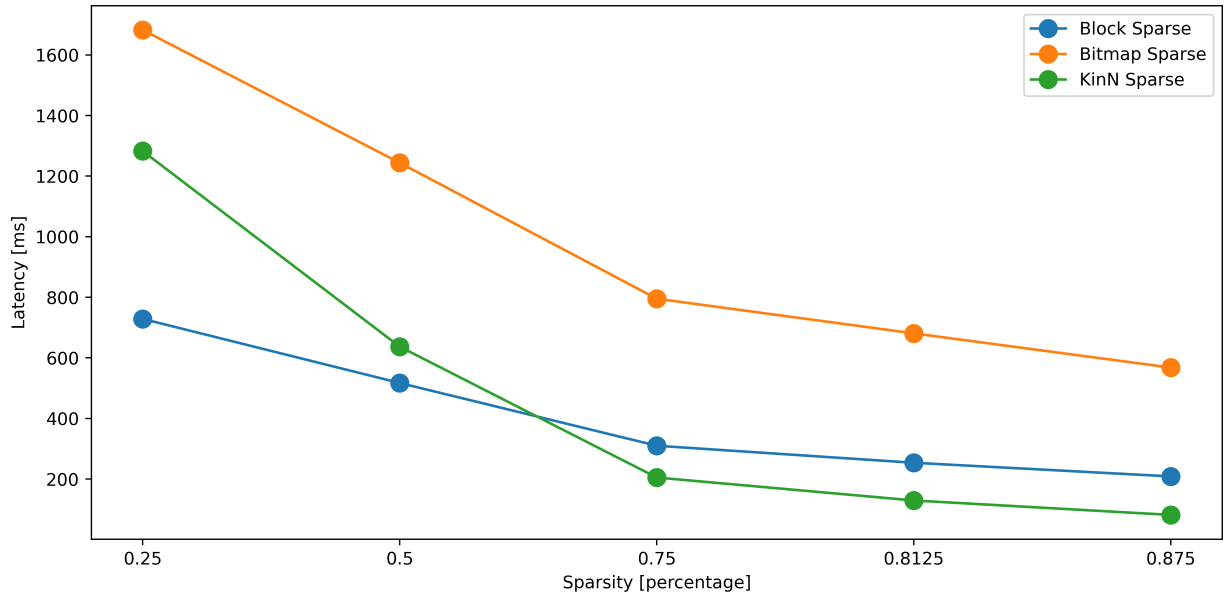

Figure 4: Square 1024 by 1024 sparse matrix dense matrix multiplication performance test on matrices with different sparsity

4 Android Phone Performance Analysis

We evaluated the performance, inference latency that is, of the K in N sparse pattern on a fully connected neural network (NN). The NN consists of four hidden layers, each composed of 1024 by 1024 sparse weight matrices and dense bias vectors with a ReLU activation function. The output layer is a 10 by 1024 weight matrix with a bias vector and a Softmax activation function. Although irrelevant for the performance analysis, the NN was trained on an upscaled MNIST data set to 32 by 32 images and the sparsity was enforced by removing the smallest absolute values of the weights. The performance was analyzed on three types of TFLite models: the default model, an optimized model, and a quantized model using 8-bit integers. These models were converted from a dense version of the NN. Moreover, we evaluated a dense implementation of the NN in NNAPI and our dense implementation using both single- and multi-threading. Lastly, we conducted performance analysis on the 4 in 16 and 2 in 16 models, again utilizing both single- and multi-threading. The 4 in 16 and 2 in 16 patterns were chosen over 1 in 4 and 1 in 8 patterns respectively, which perform just slightly better, to demonstrate more realistic conditions.
The subsequent sections outline the details of the three performance tests conducted on a Google Pixel 7 and a Xiaomi Redme Note 9 Pro phones.

4.1 Per Sample Evaluation

The per sample evaluation measures the time required to predict N samples by calling a specific model for each sample from Kotlin code. The results of these measurements are shown in Figures 5 and 6.

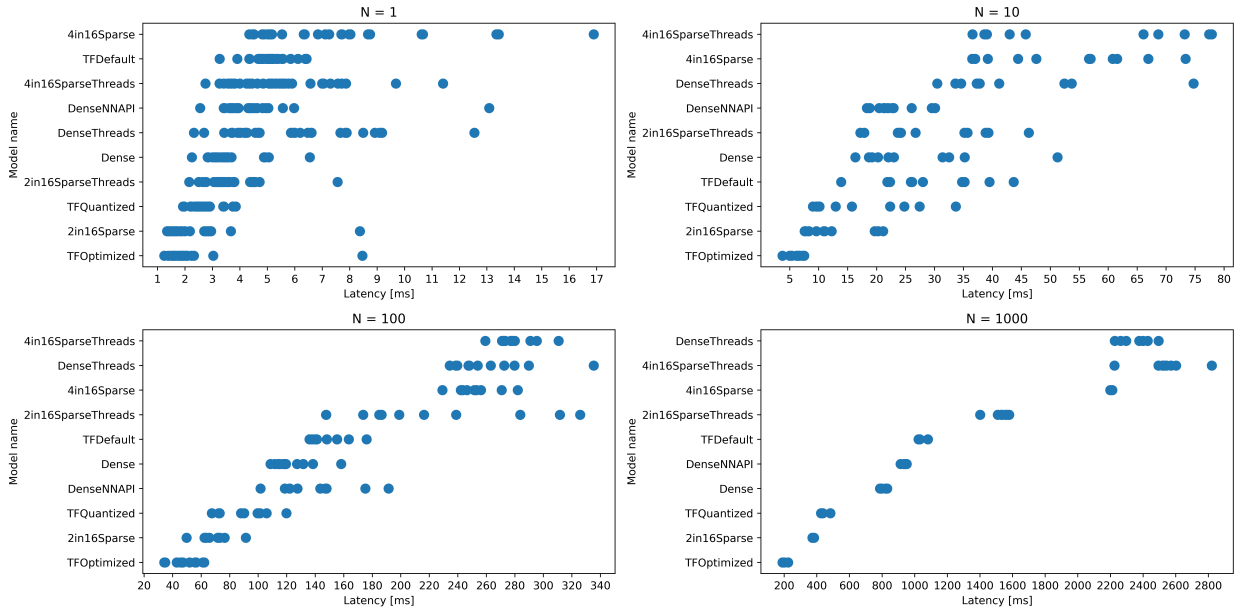

Figure 5: Per sample evaluation on Google Pixel 7

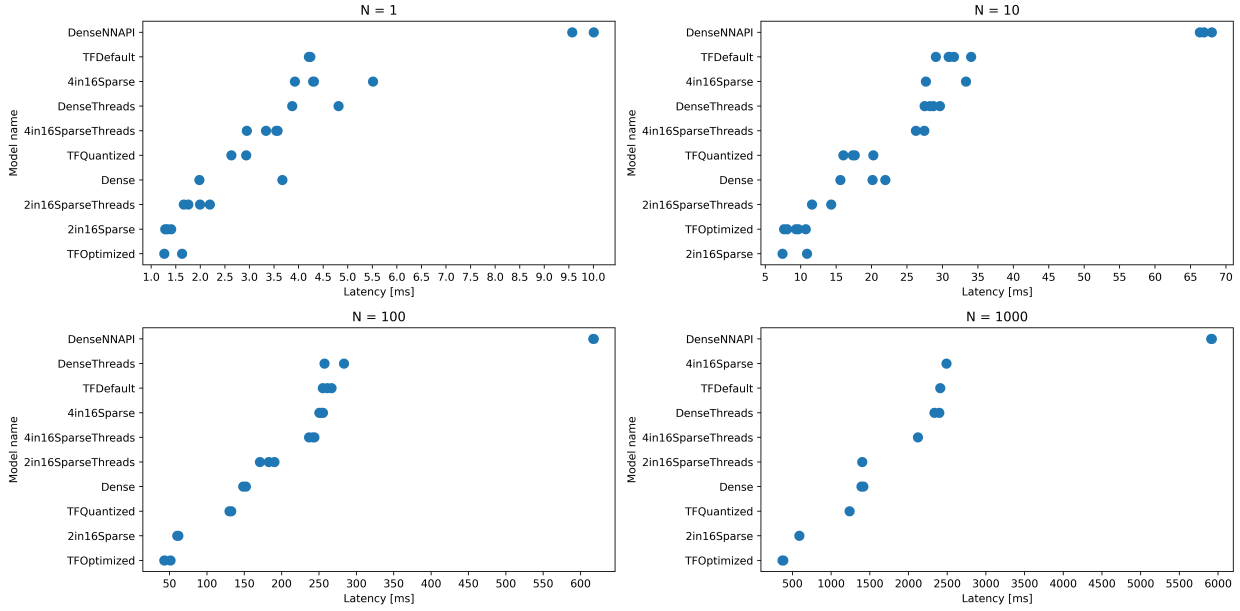

Figure 6: Per sample evaluation on Xiaomi Redme Note 9 Pro

4.2 Matrix Test Set Evaluation

This performance test measures the time required to predict the entire MNIST test set, which consists of 10 000 samples, using a single call of the native code from Kotlin. The matrix test set evaluation could not be performed on the TFLite models, as they only support per sample evaluation. The results of this evaluation are depicted in Figures 7 and 8.

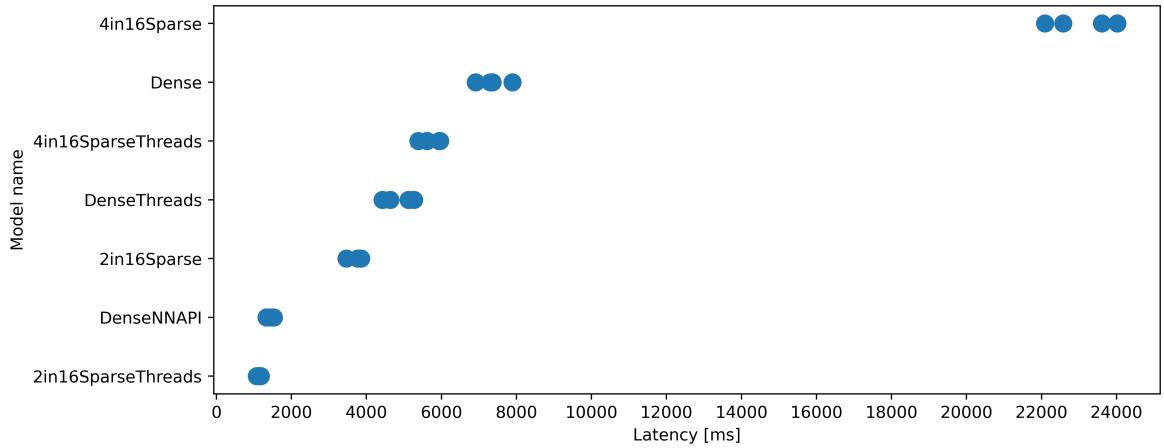

Figure 7: Matrix test set evaluation on Google Pixel 7

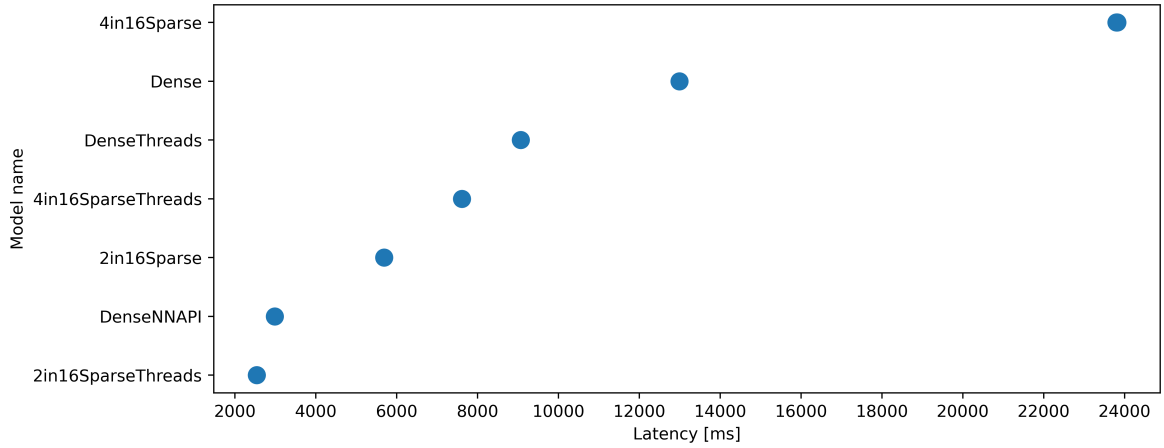

Figure 8: Matrix test set evaluation on Xiaomi Redme Note 9 Pro

4.3 Sampled Test Set Evaluation

The sampled test set evaluation measures the time required to predict the entire MNIST test set on a sample-by-sample basis. The models are called with each sample, similar to the per sample evaluation. Once again, the results of this performance analysis can be seen in Figures 9 and 10.

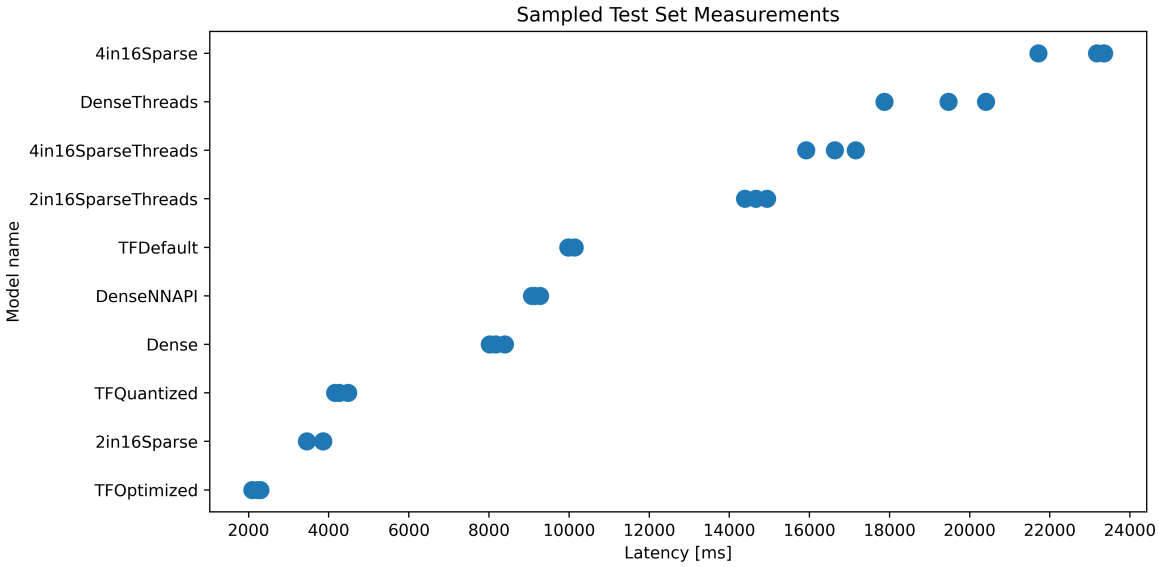

Figure 9: Sampled test set evaluation on Google Pixel 7

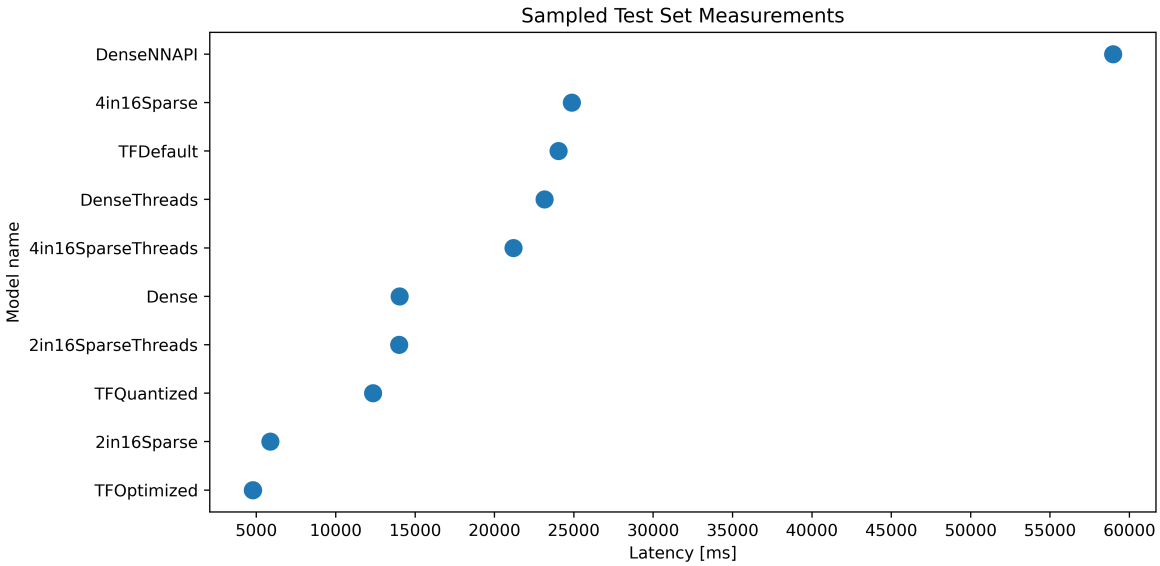

Figure 10: Sampled test set evaluation on Xiaomi Redme Note 9 Pro

5 Summary

Based on the conducted performance analysis, it is evident that the optimized TFLite model consistently outperforms even the 2 in 16 sparse model. We also explored the possibility of using a 1 in 8 sparse model, which has the same sparsity percentage but a more favorable memory layout. However, this did not yield any significant improvement, as the matrix test set evaluation only showed a minor reduction in time from 1156.29 ms to 1112.17 ms.

Furthermore, it is worth noting that the optimized TFLite model consumes less battery, indicating

that it may have access to specialized instructions. Unfortunately, these instructions are not accessible through any available API, as the NNAPI is the closest API to assembly code offered by Android. Given the great performance of the optimized TFLite model and the performance gap between the C++ implementation as well as the NNAPI implementation, there remains an open research question regarding the factors contributing to this performance gap and how it can be addressed in the C++ implementation.