# Technical Report “Vectorizing BLAS on Apple M2”

Pisit Pruangprat

---

## Introduction

Modern computer architectures often provide vector instructions to speed up programs, for example, Intel’s AVX-512 and ARM’s NEON. Those vector instructions allow us to replace multiple identical scalar instructions acting on individual data elements with a single instruction operating on multiple data elements (also known as a vector of data). For example, we can use a single instruction to add 16 pairs of integers. In order to use vector instructions, programmers need to load their data into vector registers and store it back in memory. This creates an additional data movement that is not required when working with scalar instructions and can slow down programs. However, the programs that are affected by this performance degradation are often unknown, especially when a new computer architecture, such as Apple M2, comes out.

This work is our first attempt to understand the performance of the vector instructions in the Apple M2 chip. To evaluate their performance, we pick a set of basic vector-vector operations (e.g., addition, subtraction, multiplication, and multiply-accumulate axpy) and matrix-matrix multiplication. We benchmark those operations’ vectorized implementations against their scalar implementations. We also show that the vector instructions can be further improved using a classic code optimization technique called loop unrolling. The result shows that their vectorized

implementations outperform their scalar implementations in all the workloads except the multiply-accumulate operation. In the multiply-accumulate operation, we found that the vectorized implementation is slower than the scalar implementation when the vector size is 1000.

## Experimental Setting

We use an Apple MacBook Air M2 with a 2.42 GHz CPU and 16 GB memory. Our benchmark contains six different linear algebra operations: vector-vector addition, vector-vector multiplication, vector-vector multiply-accumulate, BLAS level 1 (axpy), and BLAS level 3 (general matrix-matrix multiplication: GEMM). We implement three implementations for each operation: scalar, vectorized, and vectorized with loop unrolling (factor of 4). Except for the GEMM, we used a factor of 2 to unroll the inner loop for our vectorized implementation since our initial matrix size is too small. Our vectorized implementations use NEON intrinsic, which allows us to use ARM NEON instructions in the Apple M2 chip.

## Result

The result shows that our vectorized implementations are faster than our scalar implementations by 28.02%, 26.30%, 25.10%, 7.74%, and 58.27%, respectively, on the geometric average. This implies that vectorization using NEON intrinsic can help improve the performance of the basic linear algebra operations. An interesting insight is on the vector-vector multiply-accumulate.

We also experimented with how loop unrolling, one of the code optimization techniques, can help improve performance when using NEON. The result shows the performance improvement after applying loop unrolling by 18.99%, 18.21%, 10.58%, 19.82%, and 4.83%, respectively, on the geometric average. Note that the matrix-matrix multiplication uses 2 as the unroll factor. This also shows that even though we apply NEON intrinsic on our implementations, we can still gain performance benefits from applying loop unrolling.

Moreover, we found performance degradation when the problem size is too small or too large. We aim to unveil these in our future work.

## Conclusion

We preliminarily show that the NEON intrinsic allows programmers to improve their code performance by vectorizing their code for the Apple M2 chip. Additionally, code optimization techniques, like loop unrolling, can also be applied along with using the NEON intrinsic and further improve the performance. However, we also found some conditions, for example, too small or too large data, that the NEON intrinsic can be more inefficient than using scalar operations. We intend to explore this inefficiency in our future work.