Optimization Strategies to Accelerate BLAS Operations with ARM SVE

Aniket P. Garade, Sushil Pratap Singh, Juliya James, H V Deepika, Haribabu P, S A Kumar, S D Sudarsan
Centre for Development of Advanced Computing, Bengaluru, India
{aniketpg, sushilpratap, juliyaj, deepikahv, hari, sakumar, sds}@cdac.in

***Abstract*— Optimized mathematical libraries designed for specific hardware platforms are critical for achieving maximum performance in scientific and engineering applications. These libraries play a key role in accelerating computations and improving code efficiency. The Scalable Vector Extension (SVE) for the ARM architecture is a recent development that enhances vectorization capabilities, with wide vectors, leading to significant performance improvements. This paper explores vector optimizations for Basic Linear Algebra Subprograms (BLAS) routines, targeting both single and double precision data. It details the strategies for vectorizing BLAS operations using SVE. The approach is implemented with OpenBLAS, and experimental results reveal notable performance gains, demonstrating the efficacy of SVE in accelerating computational tasks on ARM platforms.**

***Keywords*— Math Libraries, SVE, ARM, Vectorization, BLAS, OpenBLAS, High Performance Computing**

I. Introduction

In various scientific and engineering fields, the computational capabilities of advanced processors are essential for solving complex mathematical problems. Advanced processors based on x86 and ARM architectures, utilized in servers and supercomputers, demonstrate their ability to execute computationally intensive tasks. However, to fully leverage these platforms, the choice of mathematical libraries is paramount. Math libraries serve as fundamental components in computational environments, providing a wide array of mathematical and computational functions necessary for the execution of complex numerical operations. Math libraries contain optimized routines for linear algebra, fourier transforms, differential equation solvers, and statistical functions which are exhaustively used for scientific computations. Basic linear algebra libraries [4, 5] can be categorized as Level-1 which solves vector-vector operations, Level-2 covers matrix-vector operations and Level-3 resolves matrix-matrix operations [15,16,17]. The popular implementations of BLAS operations found are ATLAS [1], GotoBLAS [2], OpenBLAS [14], Reference BLAS [20] [21], BLIS [23], Intel MKL [3] [18] and ArmPL [19].

Optimization of these libraries enhance productivity by simplifying code development, offering a broad range of operations and enabling code reusability across various hardware architectures. This efficiency in mathematical computation is crucial for advancing research and innovation, making these libraries integral to computational ecosystems.

Vectorization, is a key optimization technique, that leverages on Single Instruction Multiple Data (SIMD) type of data parallelism [6]. By executing the same operation across a data vector in parallel, vectorization reduces instruction count and maximizes processor throughput, significantly boosting performance in tasks like numerical computations and image processing. Compiler and library support, along with hardware features such as SIMD registers and execution units, are essential for effective vectorization. Different vectorization techniques include:

A. Advanced Vector Extensions

Advance Vector Extensions is a set of instructions for doing SIMD operations for x86 architectures. These instructions enhance and extend the capabilities of SIMD instruction sets, such as MMX and SSE [7,8,9]. The instruction set expanded from 256 bits AVX2 [10] to more advance AVX-512 [8], supporting 512-bit wide SIMD registers (ZMM0-ZMM31). AVX significantly enhances floating-point operations by widening the 16 XMM registers from 128 bits to 256 bits, and further extending them to 512 bits with AVX-512. It paves the way for future advancements in instruction set functionality and vector lengths through its efficient instruction encoding scheme, three and four-operand instruction syntax, and the inclusion of the Fused Multiply-Add (FMA) extension. Additionally, AVX supports load and store masking, gather operations for improved data access, and enhanced integer arithmetic. AVX features efficient broadcast instructions for replicating scalar values across vector registers, optimizing data initialization. Additionally, masked load/store operations selectively access and update memory based on conditions, enhancing efficiency in irregular data processing. These features collectively improve the computational efficiency.

B. Scalable Vector Extension

Scalable Vector Extension (SVE) is a next-generation SIMD extension for the Armv8-A aarch64 instruction set, designed to enhance the vectorization of loops that are challenging or inefficient to vectorize [11]. SVE is Vector Length Agnostic (VLA), allowing hardware implementors to choose vector register sizes that best fit their workloads [12]13]. SVE supports scalable vector lengths up to 2048 bits and includes 32 scalable vector registers (Z0-Z31) and 16 predicate registers (P0-P15). This adaptability means software can seamlessly adjust to different vector lengths without requiring new instruction encodings or recompilation.

SVE introduced several advanced features, including per-lane predication for precise control over which vector elements are active, gather-load and scatter-store for efficient data transfer from non-contiguous memory addresses, and vector partitioning for dynamic loop progression. It also supports fault-tolerant speculative vectorization, ensuring safe vector operations even with memory faults. Horizontal vector operations such as summation and logical reductions are enhanced, along with serialized vector operations that efficiently handle pointer-chasing loops, improving performance in tasks with irregular data access patterns.

The SVE programming model reduces development costs and effort in SIMD code optimization by providing a flexible approach and wider vectors, beneficial for math libraries. The VLA model ensures optimal performance across different SVE implementations, reducing deployment costs and enhancing compatibility. SVE's support for full floating-point features, compliant with the IEEE 754 standard, ensures accurate and consistent numerical computations, making it particularly advantageous for applications in scientific simulations, data analytics, and machine learning.

These advancements offer significant potential for optimizing BLAS routines. This research focuses on using the OpenBLAS library as a baseline and aims to improve the efficiency of Level 1 and Level 2 BLAS routines through optimization with SVE. The further sections of the document are classified as follows: Section 2 reviews related works, while Section 3 provides a detailed explanation of our approach used to implement SVE. Section 4 presents the experimentation and analysis. Finally, Section 5 concludes the paper, offering insights into our findings and suggesting avenues for future research.

II. Related Works

Christian Fibich, et al. [22] thoroughly evaluated various open-source BLAS libraries optimized for ARM and RISC-V architectures on Linux-capable embedded platforms. They highlighted BLAS as a critical standard for efficient linear algebra operations, crucial as IoT devices increasingly utilize powerful processors. Their findings indicate that optimized BLAS implementations offer significant performance advantages over plain C alternatives. ARM platforms demonstrate better performance compared to RISC-V platforms in their tests. This study underscores the importance of optimized libraries in enhancing computational efficiency on embedded systems and suggests future optimizations for RISC-V architecture.

Xiuwen Wan, et al. [16] presented a method to leverage ARM’s SVE and its FCMLA instruction to accelerate Level 2 BLAS routines. Their approach focused on exploiting VLA programming and wider vector capabilities introduced by SVE for the ARMv8-A architecture, aimed at enhancing vectorization efficiency. They implemented these features using simulations on ARM Instruction Emulator (ARMIE) and Gem5. Their findings demonstrate that SVE can significantly optimize Level 2 BLAS operation, such as matrix-vector multiplication.

Yi Wei, et al. [17] optimized Double Precision General Matrix Multiplication (DGEMM) on Phytium processors using ARM SVE instructions. Their approach integrates adjustments to data block sizes within the OpenBLAS library to enhance processor storage efficiency. They developed a mathematical model for determining optimal kernel sizes and implement an efficient assembly kernel to reduce memory access delays. Experimental validation underscores significant performance enhancements in DGEMM due to these optimizations with SVE instructions on Phytium processors.

III. Our Approach

This section details our ongoing implementation of SVE within the OpenBLAS-0.3.26 [24] math library. We are optimizing Level 2 BLAS routines such as Single Precision General Matrix-Vector Multiplication (SGEMV) and Double Precision General Matrix-Vector Multiplication (DGEMV). These routines are internally utilized by Single Precision Symmetric Matrix-Vector Multiplication (SSYMV) and Double Precision Symmetric Matrix-Vector Multiplication (DSYMV). For detailed optimization strategies, please refer to section (III.A). Additionally, we are addressing the optimization of Level 1 BLAS routines, including Single Precision Swapping (SSWAP), Double Precision Swapping (DSWAP), Single Precision Scaling (SSCAL), Double Precision Scaling (DSCAL), Single Precision Rotation (SROT), and Double Precision Rotation (DROT). These routines are fundamental components called internally by various BLAS operations such as symmetric and triangular matrix-vector multiplications in different formats (banded, packed, and triangular). For a thorough exploration of optimization strategies for Level 1 routines, please see sections III.B, III.C, and III.D, respectively. This research explores the implementation of SVE into BLAS routines to enhance computational performance on ARM architectures. The primary objective is to leverage SVE's scalable vectorization capabilities to improve parallelism in vector operations. The key design principles considered are

- Loop unrolling to maximize parallel iterations and exploit SVE's ability to handle multiple elements concurrently.
- Varying vector lengths allow adaptable optimization, enhancing computational efficiency and scalability in SVE-enabled operations.
- Conditional compilation to ensure compatibility with different data types such as FLOAT and DOUBLE.
- Leveraging vector load (*svld*) and store (*svst*) operations for efficient data movement, complemented by SIMD capabilities to boost computational throughput.

A. *SVE Implementation for Matrix-Vector Multiplication*

1) Algorithm Overview

- Initialization: Initialize two accumulator variables (*acc_a* and *acc_b*) to zero. These accumulators store partial sums during the computation.
- Loop Over Columns: Iterate over the columns of matrix *A* in chunks of size *sve_width * 2*. This takes advantage of SVE's dual-issue capability, allowing simultaneous processing of two segments of data per iteration. The *sve_width* is computed using *svcntw()* or *svcntd()* SVE intrinsic.
- Data Loading: Load segments of matrix *A* (*x_vec_a*, *x_vec_b*) and vector *x* (*y_vec_a, y_vec_b*) into SVE vector registers using *svld* instruction. This instruction fetch data from memory into the SVE registers, enabling vectorized operations.
- Vectorized Multiplication and Accumulation: Perform vectorized multiply-accumulate operations using *svmla_m* instruction on the loaded segments. Here, *svmla_m* multiplies each element of *x_vec_a* (or *x_vec_b*) with the corresponding element of *y_vec_a* (or *y_vec_b*) and adds the result to *acc_a* (or *acc_b*) which is shown below.

acc_a = svmla_m (pg_a, acc_a, x_vec_a, y_vec_a)

acc_b = svmla_m (pg_b, acc_b, x_vec_b, y_vec_b)

The masks *pg_a* and *pg_b* are the boolean predicates used to control which elements in the SVE vector registers are involved in the *svmla_m* operation. They ensure that only valid elements are processed, which is crucial when the data segment length does not align perfectly with the vector width (*sve_width*).

- Reduction: After completing the vectorized computation loop, use *svaddv* instruction to horizontally add the elements of *acc_a* and *acc_b*. This reduction operation consolidates the partial sums stored in the accumulator variables into a single result.
- Return Result: The result of the matrix-vector multiplication is obtained by adding the horizontal sums of *acc_a* and *acc_b*. This result represents the resulting vector y after multiplying matrix *A* with vector *x*.

2) Mathematical Expression

- Let *A* be the input matrix of size $m \times n$, and *x be* the input vector of length *n.*
- The matrix-vector multiplication operation computes the product *Ax*, where *A* is the matrix, *x* is the vector, *i* and *j* are loop indices.
- The elements of the resulting matrix-vector *Ax* are computed as:

$$(A)_{ij} = \sum_{j=0}^{n-1} A_{ij} * x_j \qquad (1)$$

In the SVE implementation, this operation is parallelized using SVE vector instructions, where multiple multiplications and additions are performed simultaneously across vector elements.

B. Implementation of SVE for Vector Swapping Operation

1) Algorithm Description

- Initialization: The function begins by taking several crucial input parameters: *n*, denoting the total number of elements in each vector, and *x* and *y*, which are pointers to the input vectors. Additionally, *inc_x* and *inc_y* represents the increments for traversing the vectors. These increments allow navigation of the vectors at specific strides, useful for handling non-contiguous memory layouts or sub-vectors within larger datasets. By accommodating these parameters, the algorithm can manage vectors of arbitrary length and stride, enhancing its versatility across various numerical computation scenarios. This initialization step sets the stage for the vectorized operations, ensuring efficient processing and swapping of elements between the two vectors.
- Vectorized Swapping Loop: The core of the algorithm is a loop that processes chunks of vector elements using SVE instructions to maximize efficiency. It begins by calculating the vector width (*sve_width*) using *svcntw()* or *svcntd()* SVE intrinsic, which return the number of 32-bit or 64-bit elements in an SVE vector, respectively, depending on whether single or double precision is used. Next, predicate (*pg*) are generated using *svwhilelt_b32* or *svwhilelt_b64* SVE intrinsic functions, which determine which elements within the current chunk are active based on their positions relative to the total number of elements (*n*). This ensures that only valid elements are processed, preventing out-of-bounds errors.

  The algorithm then loads chunks of elements from vectors *x* and *y* into SVE registers using *svld1* instruction, with the *pg* predicate ensuring that only the valid elements within bounds are loaded. Once the data is loaded, the elements from the two vectors are swapped using temporary SVE registers, which hold the data temporarily to facilitate the swap. Finally, the swapped elements are stored back into the original vectors using *svst1* instruction, again utilizing the *pg* predicate to ensure that only the valid elements are written back, respecting the bounds of the vectors. This loop iterates over the entire length of the vectors in chunks defined by the vector width, allowing for efficient and parallel processing of the swap operation.
- Loop Termination: The loop iterates over the elements in steps of the SVE vector width, terminating when all elements have been processed. This ensures that all elements are swapped efficiently, even if the total number of elements is not a multiple of the vector width.

2) Mathematical Expression

Let's denote:

- *n*: Total number of elements in each vector.
- x_i and y_i: Elements of vectors *x* and *y* at index *i*, respectively. The swapping operation can be expressed mathematically as follows:
- For $i=0, 1..., n-1$:

$$\text{Swap Operation: } x_i \leftrightarrow y_i \qquad (2)$$

C. Implementation of SVE for Vector-Scaling Operation

1) Algorithm Description

- Initialization: The initialization phase of the function sets the stage for efficient scalar multiplication using ARM SVE intrinsic. Parameters *n, x*, and *da* are crucial inputs. *n* defines the total number of elements in the vector *x*, allowing the function to iterate over the entire vector. *x* serves as a pointer to the vector's starting memory address, enabling direct access to vector elements for processing. The scalar *da* determines the value by which each element of *x* will be multiplied, a fundamental operation in scalar-vector multiplication. Together, these parameters establish the scope of operations, ensuring that the function operates on the correct data range and applies the specified scalar transformation uniformly across the vector.

- Scalar-Vector Multiplication Loop: The scalar-vector multiplication loop within the function orchestrates the efficient processing of vector elements using ARM SVE. Central to this process is a loop that iterates over the vector *x*, dividing it into chunks defined by the width of SVE vector registers (*SVE_WIDTH*). This iterative approach, starting from index *i* and incrementing by *SVE_WIDTH*, ensures that each iteration processes a contiguous block of vector elements until the entire vector *x* has been traversed.

  Within each iteration, a predicate pg is dynamically generated using *SVE_WHILELT* (*i, n*), which determines the active elements eligible for processing within the current chunk based on the iteration index *i* and the total number of elements *n.* This predicate efficiently manages memory access, ensuring that only relevant vector elements are loaded into an SVE vector register *x_vec* using *svld1*. The *pg* predicate ensures that inactive elements, beyond the vector boundary or tail end of the vector, are not accessed, thereby optimizing memory bandwidth.

  The actual scalar-vector multiplication is executed using *svmul_z*, where each element in the *x_vec* register undergoes multiplication by the scalar value *da*, producing a corresponding vector result. The predicate *pg* again plays a crucial role in this operation, ensuring that only active elements participate in the multiplication, maintaining computational efficiency.

  Finally, the modified vector result is stored back into the vector *x* using *svst1*, controlled by the same predicate *pg*. This ensures that only the elements that have been modified during the multiplication process are written back to memory, preserving data integrity and efficiency.

- Loop Termination: The loop terminates when all vector elements have been processed.

2) Mathematical Equations

Let us denote**:**

- n: Number of vector elements.
- x_i: Element of the vector x at index *i.*
- da: Scalar value used for multiplication.

The scalar-vector multiplication operation can be expressed mathematically as follows:

- For $i=0, 1..., n-1$:

$$x_i \leftarrow x_i \times da \qquad (3)$$

D. Implementation of SVE for Rotation Operation

1) Algorithm Description

The algorithm processes input vectors in chunks defined by the SVE vector width, *SVE_WIDTH*. Each iteration of the algorithm handles a segment of elements using SVE instructions. Within each iteration, a predicate *pg* is generated using the *svwhilelt* function. This predicate identifies which elements within the current chunk fall within the bounds of vectors *x* and *y*. This approach ensures precise handling of boundary conditions, particularly when the number of elements isn't a perfect multiple of *SVE_WIDTH*. By leveraging SVE's capabilities in this manner, the algorithm maximizes efficiency in processing vector operations on ARM architectures.

The code initiates by utilizing the *svld1* instruction to load pairs of elements from vectors *x* and *y* into SVE vectors *x_vec* and *y_vec*. This loading process is optimized by *svld1's* capability to leverage the active predicate, which selectively identifies elements within the vectors that are actively processed during the load operation. This approach enhances the parallel data handling capabilities of the processor.

Once the elements are successfully loaded into *x_vec* and *y_vec*, the algorithm proceeds to perform a given rotation using the given *cosine* (*c*) and *sine* (*s*) values. This rotation operation modifies the elements within *x_vec* and *y_vec* according to the defined transformation, ensuring efficient computation of the desired transformation across the vector elements. The algorithm calculates *cx_vec* as the product of *x_vec* and *c*, and *sy*_vec as the product of *y_vec* and *s* using *svmul_z* intrinsic. These are added using *svadd_z* to form the new *x* elements. Similarly, *sx_vec* (product of *x_vec* and *s*) and *cy_vec* (product of *y_vec* and *c*) are subtracted to form the new *y* elements. The updated elements are stored back into the vectors *x* and *y* using *svst1.*

2) Mathematical Expression

Mathematically, the given rotation operation can be expressed as follows: Given elements x_i and y_i from vectors x and y respectively, and rotation parameters *c* (*cosine*) and *s* (*sine*), the rotation operation transforms the elements as:

$$x_i' = c \cdot x_i + s \cdot y_i \qquad (4)$$

$$y_i' = c \cdot y_i - s \cdot x_i \qquad (5)$$

where x_i' and y_i' represent the updated elements after rotation.

IV. Experimentation And Analysis

A. Experimental Setup

To evaluate the effect of our optimizations on the BLAS library, we assess its performance on two different variants of A64FX processors, which notably differ in clock speed and core count. These processors are integral components within a larger cluster environment of PARAM Neel from C-DAC, India & Fugaku supercomputer from RIKEN, Japan.

TABLE I. PLATFORMS AND SPECIFICATIONS

Info	PARAM Neel	Fugaku
Processor	A64FX	A64FX
Clock speed	1.8GHz	2.2GHz
Cores	48	52
Vectorization	SVE	SVE
Memory	32 GB HBM2	32 GB HBM2
Compiler	GCC-12.2	GCC-12.2

Table 1. specifies the details of the hardware capabilities of the processors used.

B. Observation and Analysis

We compare our SVE optimized OpenBLAS v0.3.26 routines with the original OpenBLAS v0.3.26 routines on PARAM Neel and Fugaku.

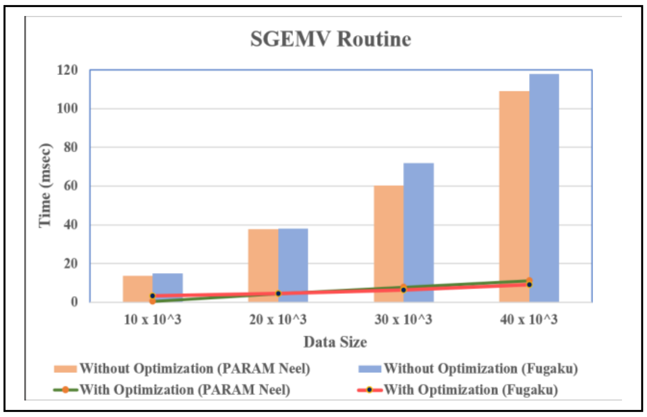

Fig. 1. Performance of SGEMV Routine

Fig. 1 shows the comprehensive performance analysis of SGEMV BLAS routine. In general, the SGEMV routine achieves performance gain on an average by approximately 12.73x on PARAM Neel and 8.54x on Fugaku. For smaller data size, the performance of optimized BLAS on PARAM Neel is approximately 6.14x times better as compared to optimized BLAS on Fugaku. As the data size increases the optimized BLAS on Fugaku shows performance gain of approximately 1.15x times better than the optimized BLAS on PARAM Neel.

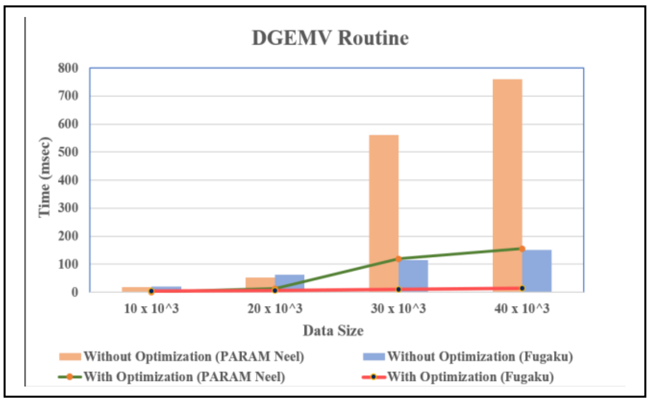

Fig. 2. Performance of DGEMV Routine

Fig. 2 illustrates the performance analysis of DGEMV BLAS routine. For smaller datasets, the optimized BLAS routines exhibit an average performance improvement of 10.27x on PARAM Neel and 7.58x on Fugaku. However, as the data size increases, the performance scaling on PARAM Neel does not maintain the same efficiency observed on Fugaku, leading to an average performance enhancement of 10.88x on Fugaku compared to 4.80x on PARAM Neel. On the initial datasets, we observe that the performance of optimized BLAS is almost similar on both the platforms. However, as the dataset size increases, the performance scaling of optimized BLAS on PARAM Neel reduced and is comparable to the unoptimized performance of Fugaku.

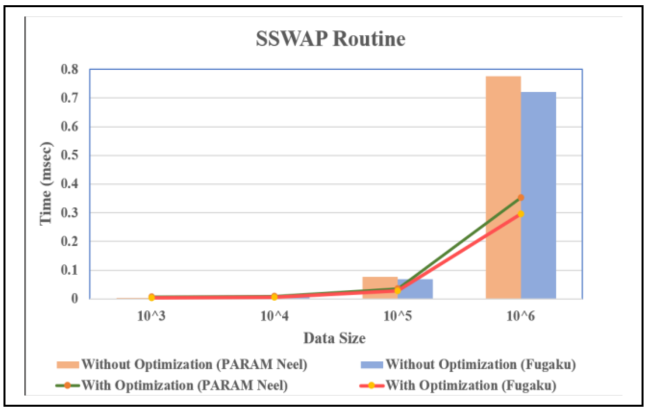

Fig. 3. Performance of SSWAP Routine

Fig. 3 depicts the performance analysis of SSWAP BLAS routine. For smaller datasets, the performance of the optimized BLAS routines is nearly identical to the routine without SVE optimizations on both PARAM Neel and Fugaku. For the larger date size of 1 million elements, the SSWAP routine achieves performance improvement of 2.20x and 2.43x on PARAM Neel and Fugaku respectively.

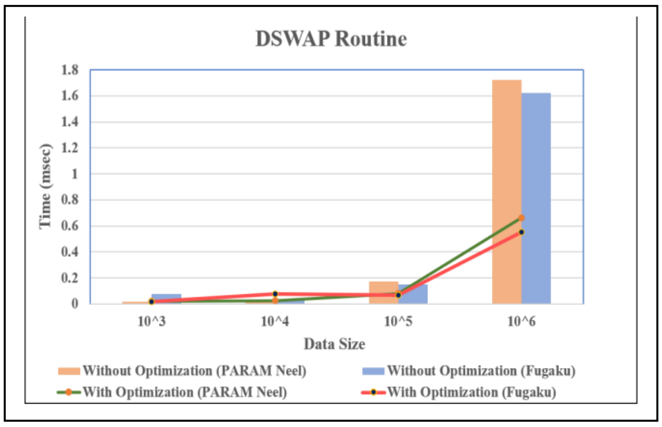

Fig. 4. Performance of DSWAP Routine

Fig. 4 compares the SVE performance of DSWAP BLAS routine. For smaller datasets, the performance of both the optimized and original BLAS routine is similar on both PARAM Neel and Fugaku, with an observed spike in performance at a sample size of 10^4 on Fugaku. For larger datasets, such as 1 million elements, the DSWAP routine achieves a performance improvement of 2.60x on PARAM Neel and 3.00x on Fugaku.

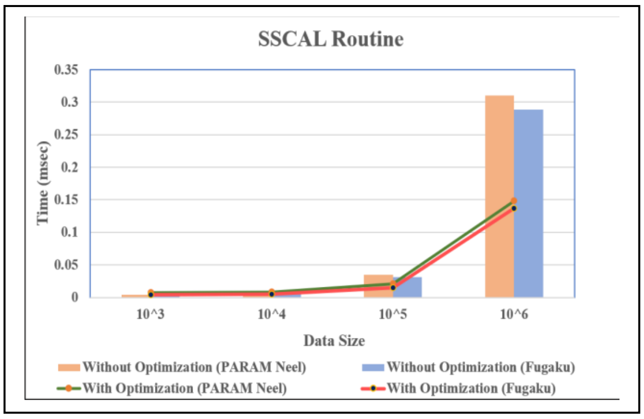

Fig. 5. Performance of SSCAL Routine

Fig. 5 shows the performance analysis of SSCAL BLAS routine. For smaller datasets, the performance of optimized BLAS is almost similar to original BLAS on both PARAM Neel and Fugaku. For larger data size of 1 million elements, the SSCAL routine achieves performance gain of 2.10x on both the platforms.

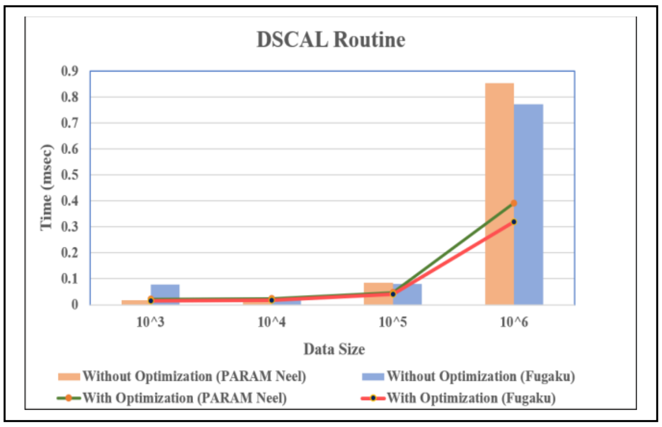

Fig. 6. Performance of DSCAL Routine

Fig. 6 depicts the performance analysis of the DSCAL BLAS routine. For smaller datasets, the performance of the optimized BLAS is identical to that of the original BLAS. However, for larger datasets, specifically those with 1 million elements, the DSCAL routine achieves a performance improvement of 2.17x on PARAM Neel and 2.41x on Fugaku.

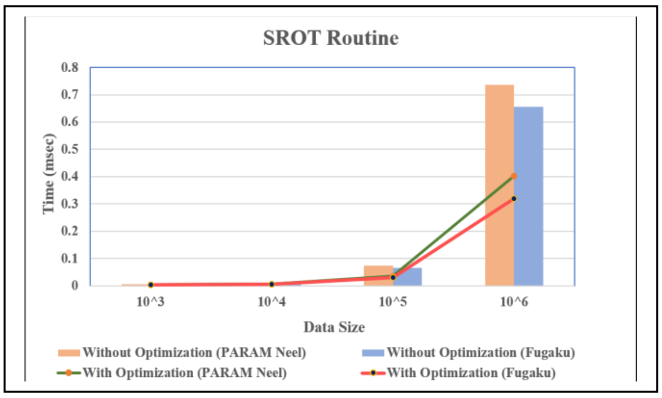

Fig. 7. Performance of SROT Routine

Fig. 7 shows the performance analysis of SROT BLAS routine. For lower datasets, the performance of optimized BLAS is identical to without SVE optimization on both PARAM Neel and Fugaku. However, for larger datasets of 1 million elements, the SROT routine achieves a performance gain of 1.83x on PARAM Neel and 2.03x on Fugaku.

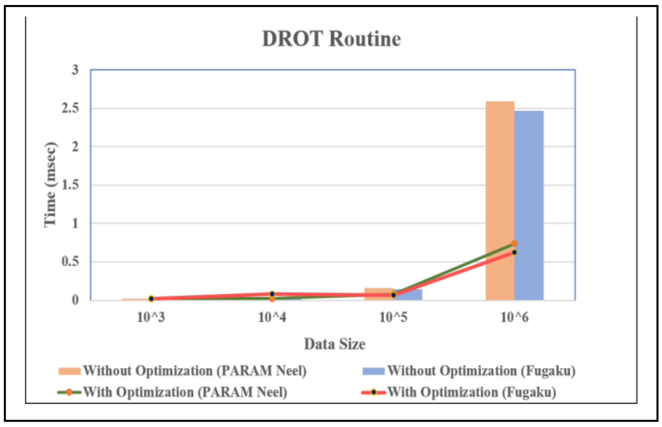

Fig. 8. Performance of DROT Routine

Fig. 8 illustrates the performance analysis of the DROT BLAS routine. For smaller datasets, the performance of the optimized BLAS is almost identical to the original BLAS routine on both PARAM Neel and Fugaku. However, for larger datasets of 1 million elements, the DROT routine achieves a performance improvement of 3.50x on PARAM Neel and 3.96x on Fugaku.

From the above analysis, it is observed that the performanace of double precision optimized BLAS routines is better when compared to single precision optimized BLAS routines.

V. Conclusion And Future Work

SVE introduces new architectural features that provide wider vectors and enable the vectorization of loops on ARM platforms. In this paper, we proposed an optimized implementation of Level 1 and Level 2 BLAS routines in OpenBLAS library. The effect of SVE based optimization on OpenBLAS is demonstrated and analyzed with two different variants of ARM processors. The results indicate significant performance improvement due to effective code vectorization, resulting in enhancements ranging from 7x to 13x for Level 2 routines and 1.80x to 4x for Level 1 routines. As part of future research, we intend to implement additional BLAS routines leveraging SVE optimizations.

Acknowledgment

This work utilized the computational resources of the supercomputer Fugaku provided by RIKEN. The paper is a research work under the National Supercomputing Mission Project, India. We would like extend our gratitude to Himanshu Chaudhary and Saurav Ravindran without whom this work would not have been possible.

References

[1] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In Supercomputing (SC), pages 1–27, 1998

[2] K. Goto and R. A. v. d. Geijn. Anatomy of high performance matrix multiplication. ACM Transactions on Mathematical Software (TOMS), 34(3):12:1–12:25, 2008.

[3] Intel. Intel math kernel library (MKL). http:// software.intel.com/en-us/intel-mkl.

[4] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. ACM Transactions on Mathe matical Software (TOMS),14(1):1–17,1988.

[5] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Du,"A set of level 3 basic linear algebra subprograms," ACMTransactions on Mathematical Software (TOMS), 16(1):1–17, 1990.

[6] L. Huang, S. Ma, L. Shen, Z. Wang, and N. Xiao, "Low-cost binary128 floating-point fma unit design with simd support," IEEE Transactions on Computers, vol. 61, no. 5, pp. 745–751, 2011

[7] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer Manuals. Retrieved November 11, 2019 from https://software.intel.com/en-us/articles/ intel-sdm

[8] Intel. 2019. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. https://software.intel.com/en-us/download/ intel-64-and-ia-32-architectures-software-developers-manual-volume-1-basic-\ architecture

[9] Daniel S. McFarlin, Volodymyr Arbatov, Franz Franchetti, and Markus Püschel. 2011. Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets. In Proceedings of the International Conference on Super computing (ICS'11). Association for Computing Machinery, New York, NY, USA, 265–274. https://doi.org/10.1145/1995896.1995938

[10] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. IEEE Micro 36, 2 (Mar 2016), 34–46. https://doi.org/10.1109/MM.2016.25

[11] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu et al., "The arm scalable vector extension," IEEE Micro, vol. 37, no. 2, pp. 26–39, 2017.

[12] A. Pohl, M. Greese, B. Cosenza, and B. Juurlink, "A performance analysis of vector length agnostic code," in Proceedings of the 2018 InternationalConferenceonHighPerformanceComputing&Simulation (HPCS), 2019

[13] J. Lee, F. Petrogalli, G. Hunter, and M. Sato, "Extending openmp simd support for target specific code and application to arm sve," in International Workshop on OpenMP. Springer, 2017, pp. 62–74.

[14] Openblas : Z. Xianyi, W. Qian, and W. Saar, "Openblas: An optimized blas library," Accedido: Agosto, 2016

[15] Q. Wang, X. Zhang, Y. Zhang and Q. Yi, "AUGEM: Automatically generate high performance Dense Linear Algebra kernels on x86 CPUs," SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 2013, pp. 1-12, doi: 10.1145/2503210.2503219

[16] X. Wan, N. Gu and J. Su, "Accelerating Level 2 BLAS Based on ARM SVE," 2021 4th International Conference on Advanced Electronic Materials, Computers and Software Engineering (AEMCSE), Changsha, China, 2021, pp. 1018-1022, doi: 10.1109/AEMCSE51986.2021.00208.

[17] Y. Wei, L. Deng, S. Sun, S. Li and L. Shen, "DGEMM Optimization Oriented to ARM SVE Instruction Set Architecture," 2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS), Nanjing, China, 2023, pp. 514-521, doi: 10.1109/ICPADS56603.2022.00073.

[18] R. Lim, Y. Lee, R. Kim, and J. Choi, "An implementation of matrix–matrix multiplication on the intel knl processor with avx-512," Cluster Computing, vol. 21, no. 4, pp. 1785–1795, 2018.

[19] "ArmPL", [online], Available: https://developer.arm.com/, June 2024

[20] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry et al., "An updated set of basic linear algebra subprograms (blas)," ACM Transactions on Mathematical Software, vol. 28, no. 2, pp. 135–151, 2002.

[21] BLAST Forum, "Basic Linear Algebra Subprograms Technical Forum Standard," https://netlib.org/blas/blast-forum/blas-report.pdf, 2020-06-27, University of Tennessee, Knoxville, Tennessee, Tech. Rep., 2001

[22] C. Fibich, S. Tauner, P. Rössler and M. Horauer, "Evaluation of Open-Source Linear Algebra Libraries targeting ARM and RISC-V Architectures," 2020 15th Conference on Computer Science and Information Systems (FedCSIS), Sofia, Bulgaria, 2020, pp. 663-672, doi: 10.15439/2020F145.

[23] F. G. Van Zee and R. A. Van De Geijn, "Blis: A framework for rapidly instantiating blas functionality," ACM Transactions on Mathematical Software (TOMS), vol. 41, no. 3, pp. 1–33, 2015.

[24] https://github.com/OpenMathLib/OpenBLAS/releases/tag/v0.3.26