# oneDAL Optimization for ARM Scalable Vector Extension: Maximizing Efficiency for High-Performance Data Science

Chandan Sharma, Rakshith GB, Ajay Kumar Patel, Dhanus M Lal, Darshan Patel, Ragesh Hajela, Masahiro Doteguchi, Priyanka Sharma Fujitsu Research of India Pvt. Ltd.

Email: {chandan.sharma, rakshith.gb, ajaykumar.patel, dhanus.mlal, darshan.patel, ragesh.hajela, masahiro.doteguchi, priyanka.s}@fujitsu.com

Abstract—The evolution of ARM-based architectures, particularly those incorporating Scalable Vector Extension (SVE), has introduced transformative opportunities for high-performance computing (HPC) and machine learning (ML) workloads. The Unified Acceleration Foundation's (UXL's) oneAPI Data Analytics Library (oneDAL) is a widely adopted library for accelerating ML and data analytics workflows, but its reliance on Intel®'s proprietary Math Kernel Library (MKL) has traditionally limited its compatibility to x86 platforms. This paper details the porting of oneDAL to ARM architectures with SVE support, using OpenBLAS as an alternative backend to overcome architectural and performance challenges.

Beyond porting, the research introduces novel ARM-specific optimizations, including custom sparse matrix routines, vectorized statistical functions, and a Scalable Vector Extension (SVE)-optimized Support Vector Machine (SVM) algorithm. The SVM enhancements leverage SVE's flexible vector lengths and predicate-driven execution, achieving notable performance gains 22% for the Boser method and 5% for the Thunder method. Benchmarks conducted on ARM SVE-enabled AWS Graviton3 instances showcase up to 200X acceleration in ML training and inference tasks compared to the original scikit-learn implementation on ARM platform.

Moreover, the ARM-optimized oneDAL achieves performance parity with, and in some cases exceeds, the x86 oneDAL implementation (MKL backend) on Ice Lake x86 systems, which are nearly twice as costly as AWS Graviton3 ARM instances. These findings highlight ARM's potential as a high-performance, energy-efficient platform for data-intensive ML applications. By expanding cross-architecture compatibility and contributing to the open-source ecosystem, this work reinforces ARM's position as a competitive alternative in the HPC and ML domains, paving the way for future advancements in data-intensive computing.

Index Terms—High-Performance Computing (HPC), ARM Scalable Vector Extension (SVE), oneAPI Data Analytics Library (oneDAL), Math Kernel Library (MKL), Vector Statistical Library (VSL), OpenRNG, OpenBLAS, Machine Learning (ML), Data Analytics, Performance Optimization, ARM Architecture.

#### I. Introduction

The growing power of computing fuels scientific discovery and technical advancements, enabling enhanced applications in machine learning and big data analytics, with high-performance computing (HPC) systems depending on efficient hardware and software designs to drive these improvements [1]. X86-based architectures have historically

dominated the HPC environment because of their software availability, robust performance, and wide ecosystem support [3]. However, this landscape is being redefined by the latest developments in ARM-based processors, such as the A64FX and FX700[4]. Both the FX700 and the A64FX, which are powerful candidates for next-generation HPC systems because of their exceptional performance and energy efficiency, are employed in the Fugaku supercomputer[5]. The upcoming ARM processors[6], [7], [8], [9], which foresee advancements in the future and make use of complex microarchitecture to give improved performance and energy efficiency, further this trend. Energy-efficient computing is crucial to meet the increasing demands of data centers, and ARM CPUs represent a significant advancement in this area. This idea is aligns with global environmental objectives, aiming to significantly reduce energy consumption in data centers [10].

Among the significant contributions to the HPC and ML ecosystem, the oneAPI Data Analytics Library (oneDAL) [11], developed under the UXL [12]. oneDAL provides highly optimized routines for data preprocessing, transformation, analysis, modeling, and validation, leveraging hardware acceleration techniques such as parallelism and vectorized computations. Widely integrated into Python-based workflows through daal4py [13], oneDAL enables seamless adoption within popular data science and machine learning framework such as scikit-learn [14]. However, oneDAL's reliance on MKL [15] restricts its applicability to x86 platforms [16], creating challenges for adoption on ARM systems.

As ARM processors continue to gain prominence in the HPC landscape, adapting oneDAL for ARM platforms is essential for achieving high-performance data analytics across a broader spectrum of hardware. ARM SVE introduces architectural innovations that overcome limitations inherent in traditional SIMD (Single Instruction, Multiple Data) architectures, such as Intel's AVX [17] and ARM's NEON [18]. Unlike fixed-width SIMD systems, SVE employs a Vector Length Agnostic (VLA) approach, allowing vector lengths to scale flexibly between 128 and 2048 bits in increments of 128 bits. This adaptability enables a single, vectorized implementation to operate efficiently across different ARM processors without

architecture-specific modifications, making SVE particularly suitable for scientific and data analytics applications that demand high parallelism and scalability.

A defining characteristic of Scalable Vector Extension (SVE) [19] is that it is a separate extension from Advanced SIMD (commonly known as Neon), with an entirely new set of instruction encodings specifically designed to address the needs of high-performance computing (HPC). Unlike Neon, which is more general-purpose, SVE is tailored to overcome traditional barriers to auto-vectorization in compilers, enabling more efficient and scalable parallel processing. SVE introduces several key features that significantly enhance flexibility and performance in vectorized computations, making it particularly suited for data-intensive applications such as machine learning and scientific computing, where complex and irregular data patterns are common. These features include:

- Gather-load and scatter-store instructions for efficient memory access patterns.
- **Per-lane predication**, enabling independent operations on different vector elements.
- Predicate-driven loop control and management, providing fine-grained control over iterations in vectorized loops.

To implement its advanced capabilities, SVE introduces two new types of registers designed to enhance flexibility and efficiency in vector processing:

**Z Registers:** SVE provides 32 Z registers, which are configurable in width and can hold data elements interpreted as 8-bit, 16-bit, 32-bit, or 64-bit values. Each Z register's width can range up to 2048 bits, with a 2048-bit Z register capable of holding 256 8-bit elements, 128 16-bit elements, 64 32-bit elements, or 32 64-bit elements. This adaptability allows Z registers to handle large datasets in parallel, optimizing performance for data-intensive applications, such as ML algorithms, where multiple elements are processed simultaneously.

**P Registers:** SVE also introduces 15 predicate registers (P registers) that enable fine-grained control over vector operations by acting as masks. P registers serve the following purposes:

- **P0–P7**: Used as governing predicates for load/store and arithmetic operations.
- P8-P15: Serve additional roles in loop management and control.
- FFR (First Fault Register): Provides hardware support for speculative execution and assists in fault recovery during vectorized operations.

The flexibility provided by SVE's Vector Length Agnostic (VLA) processing model, combined with the selective computation capabilities enabled by P registers, makes SVE particularly suited for HPC and ML applications.

Adapting oneDAL for ARM's SVE architecture involves several challenges. First, a new backend needs to replace MKL, as it is incompatible with ARM systems. OpenBLAS [20], an open-source alternative, provides fundamental BLAS and LAPACK functionalities for ARM, but lacks the advanced

optimizations that MKL offers on x86 platforms. Thus, specific ARM-compatible implementations and optimizations are necessary to achieve comparable performance. Second, to fully leverage ARM SVE, key components of oneDAL must be re-engineered to exploit SVE's unique capabilities, including the vectorized ML algorithm, optimized statistical functions, and custom sparse matrix routines. By implementing ARM-specific enhancements oneDAL can achieve efficient computation for ML tasks on ARM platforms.

#### II. RELATED WORK

The oneAPI initiative provides a unified programming model designed to support diverse hardware architectures, including CPUs, GPUs, and FPGAs. As part of this ecosystem, oneDNN has already been successfully ported to ARM, demonstrating substantial performance improvements on ARM-based platforms [21]. This success paved the way for the adaptation of the oneAPI Data Analytics Library (oneDAL) to ARM, allowing it to leverage ARM's advanced architectures for high-performance data analytics while broadening its compatibility beyond x86 and MKL systems.

Original scikit-learn pipelines include libraries like NumPy [22] and SciPy [23], which utilize OpenBLAS for numerical computations. *daal4py* acts as a bridge between scikit-learn and oneDAL, enabling accelerated machine learning (ML) algorithms through the scikit-learn-intelex patch [24]. However *daal4py* heavily relies on MKL for its performance optimizations, limiting its applicability to x86 platforms.

The oneAPI initiative also includes SYCL, a cross-platform parallel programming model designed to manage heterogeneous hardware. SYCL facilitates portability across GPUs, CPUs, and other accelerators, offering a unified framework for efficient execution [25]. However, the implementation of SYCL in oneDAL currently requires a compatible compiler, such as Intel's DPC++ [26], which is outside the scope of this research. This work focuses on extending oneDAL for ARM architectures, specifically using OpenBLAS and ARM SVE. Despite this, researchers continues to monitor developments in SYCL and DPC++ within the Unified Acceleration Foundation (UXL) to explore potential future integration.

Efforts to replace MKL with alternatives like OpenBLAS have gained traction to broaden oneDAL's compatibility with ARM architectures. OpenBLAS provides cross-platform functionality with optimizations for processors such as ARM, RISC-V, sandybridge [27] and Loongson [28]. However, OpenBLAS lacks advanced modules like Sparse BLAS (SPBLAS) and the Vector Statistical Library (VSL) offered by MKL, which are essential for high-performance ML workloads (see Figure 1) [29], [30]. These developments hold the potential to further enhance oneDAL's high-performance capabilities across diverse hardware ecosystems.

MKL's advanced capabilities, including SPBLAS for sparse matrix operations and VSL for statistical computations, underpin its exceptional performance in x86-based ML applications. VSL supports a wide range of pseudo-random number generators (RNG) and vectorized statistical routines, critical for

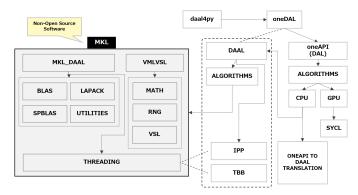


Fig. 1: Dependencies of oneDAL on MKL

ML tasks. Additionally, MKL ensures efficient multi-threaded performance with libraries like IPP (Integrated Performance Primitives) and TBB (Threading Building Blocks) [31], which facilitate large-scale parallel computations. In contrast, while OpenBLAS offers essential BLAS and LAPACK functionalities, it lacks the advanced capabilities of SPBLAS and VSL. As a result, OpenBLAS often lags behind MKL in tasks such as sparse matrix operations and compact GEMM (General Matrix Multiply) operations. Despite its open-source flexibility, OpenBLAS performance for these advanced tasks is not yet on par with MKL. However, ongoing developments, such as the addition of support for Tall-and-Skinny Matrix Multiplication (TSMM) [32], are helping to bridge this gap and improve its applicability in deep learning. These developments hold the promise of further enhancing oneDAL's performance across diverse hardware ecosystems.

Sparse Matrix Processing for ARM Architectures Sparse matrix operations are fundamental in machine learning (ML) workloads, especially when processing large datasets with sparse structures containing numerous zero elements. Efficient handling of sparse matrices is critical for algorithms such as Principal Component Analysis (PCA), KMeans clustering, and linear regression, which depend heavily on matrix-vector and matrix-matrix operations. Traditionally, x86 platforms benefit from oneAPI Math Kernel Library, which includes the Sparse BLAS (SPBLAS) module—a robust collection of routines tailored to accelerate these computations. MKL's SPBLAS [33] module provides a comprehensive set of operations for sparse matrices and vectors, organized into four main groups:

- State Management Routines: These routines initialize, configure, and manage data structures, such as sparse::matrix\_handle\_t, which encapsulates sparse matrix representations.
- Analysis Routines: Often referred to as the "inspector" or "optimize" stage, these routines inspect matrix properties (e.g., size, sparsity pattern, and parallelism). They create optimized internal data structures or apply transformations to enable efficient execution, without altering user-provided data.
- Execution Routines: These perform the actual matrixvector and matrix-matrix operations, leveraging optimiza-

- tions and metadata from the analysis stage.
- Helper Routines: These utilities handle data manipulation tasks, such as transposing matrices, copying data between handles, or reformatting sparse matrices.

This modular and highly optimized structure has enabled x86 systems to efficiently process sparse data, achieving significant performance gains in ML and data analytics workloads. However, OpenBLAS, a widely used alternative to MKL for non-x86 platforms, does not offer equivalent support for sparse computations. This creates a performance gap when running sparse matrix operations on ARM systems. To bridge this gap, our work focuses on porting and optimizing sparse BLAS routines for ARM architectures within the oneDAL framework. These efforts involve developing tailored optimizations to efficiently process sparse matrices in formats such as Compressed Sparse Row (CSR).

Vector Statistical Library (VSL) for ARM Architectures Statistical functions are critical to numerous machine learning and data analysis workflows, supporting tasks such as feature scaling, dimensionality reduction, and model evaluation. MKL includes the VSL, which provides highly optimized implementations for x86 platforms. VSL offers advanced functionalities such as random number generation, convolution and correlation, and summary statistics calculations, all structured around task objects data descriptors that encapsulate operation parameters, enabling efficient computation and reusability[34]. These functions are vital for handling large-scale datasets and computationally intensive ML workflows, including variance calculations, cross-products, and covariance matrices. However, ARM platforms have historically lacked equivalent support, creating performance bottlenecks in statistical computations. To address this gap, this work adapts two key VSL routines, variance calculation x2c\_mom and cross-product computation xcp for ARM architectures. These operations are essential in preprocessing and ML algorithms such as PCA and linear regression.

# Random Number Generation (RNG) for ARM Architectures

Random Number Generation plays a crucial role in many machine learning algorithms. In x86-based workflows, oneDAL leverages MKL Vector Statistical Library RNG, which supports a wide range of random number engines and distributions. This includes pseudorandom, quasi-random, and non-deterministic random number generation, providing efficient and scalable performance on x86 platforms. However, on ARM platforms, RNG functionality has been limited to the standard C++ RNG backend, which offers only basic engines, such as MT19937 (Mersenne Twister). This limitation has hindered the performance and versatility of ARM-based ML workflows, particularly for large-scale and parallelized applications.

To overcome this limitation, OpenRNG was integrated as the RNG backend for ARM optimized oneDAL. OpenRNG, an open-source library, was originally released with Arm Performance Libraries 24.04 [35] and is designed as a replacement for MKL VSL RNG. It enhances the RNG capabilities on ARM by supporting advanced engines like MT19937 and MCG59 (Multiplicative Congruential Generator). OpenRNG is engineered to maximize performance in multi-threaded and distributed environments, providing three parallel generation methods:

- Family Method: Generates independent streams of random numbers across threads.
- SkipAhead Method: Skips forward in the random sequence to generate disjoint subsets.
- 3) **LeapFrog Method**: Allocates alternating elements of the sequence to different threads for parallelism.

OpenRNG's integration has been shown to provide substantial performance improvements for other experiments such as 44x speedup for PyTorch's dropout layer and 2.7x speedup over the C++ standard library RNG [36]. This makes OpenRNG a key enabler for accelerating machine learning workflows on ARM platforms, particularly in applications requiring high-quality randomness, such as stochastic modeling and AI-driven simulations.

SVM Algorithms for ARM SVE Architectures Support Vector Machines (SVMs) are widely employed in machine learning (ML) tasks, including classification and regression. In this work, the SVM implementation within oneDAL has been optimized for ARM SVE, enhancing computational throughput and enabling efficient parallel execution. Initially, the SVM performance on ARM was comparable to or worse than its x86 using the MKL backend counterpart. This performance gap prompted a deeper analysis of the implementation to optimize and ensure that the ARM SVE optimized oneDAL could match or exceed the performance of default oneDAL on x86 with MKL. Upon further investigation, we identified key computational bottlenecks that could be alleviated through fine-tuning and optimization using SVE's unique capabilities. Key computational processes within SVM, such as working set selection (WSS) and Lagrange multiplier updates, were re-engineered to leverage SVE's features. The vector-length agnostic (VLA) processing of SVE allows dynamic adjustment to varying hardware configurations, while predicate registers ensure efficient handling of conditional computations by enabling selective data processing within vectorized instructions.

#### III. CONTRIBUTIONS

This work makes several important contributions to the adaptation and optimization of the oneDAL library for ARM processors:

- The oneDAL was successfully ported to ARM SVE and NEON architecture by utilizing OpenBLAS alternative of MKL, enabling high-performance machine learning and data analytics on ARM-based systems while maintaining compatibility with x86 architectures.
- 2) Efficient sparse matrix operations were implemented within oneDAL, specifically designed for compressed sparse row (CSR) format data. The newly developed routines, including csrmm, csrmultd, and csrmv, enable oneDAL to perform sparse matrix computations effectively on ARM processors.

- 3) A novel implementation of the Vector Statistical Library (VSL) was developed, enabling statistical routines, such as variance calculations and cross-product matrix computations, to be executed efficiently on ARM hardware.
- 4) OpenRNG was integrated as the backend random number generator for oneDAL, replacing the default stdc++ implementation on non-x86 platforms. This integration extended the RNG functionalities available in oneDAL, opening up more optimization opportunities.
- 5) The Support Vector Machine (SVM) algorithm was optimized, with a specific focus on the Working Set Selection (WSS) function, to take advantage of ARM's SVE features. By utilizing SVE's vectorization capabilities, significant performance improvements were achieved on ARM-SVE based systems.

The optimizations and validations implemented are publicly available as part of the *oneDAL* open-source project on GitHub [37], allowing the community to use these enhancements for ARM support.

#### IV. METHODOLOGY AND IMPLEMENTATION

This section presents a structured approach to enabling *oneDAL* for ARM architectures, with a focus on ARM's Scalable Vector Extension (SVE) optimizations. Key areas covered include ARM enablement, Sparse BLAS optimizations, Vector Statistical Library (VSL) implementations, Support Vector Machine (SVM) optimizations with Scalable Vector Extension, and Random Number Generator (OpenRNG) integration. Each subsection is demonstrating how these optimizations collectively enhance *oneDAL's* on ARM SVE Architecture.

## A. Optimizing oneDAL for ARM SVE Compatibility

The primary objective of optimization is to adapt *oneDAL* to work efficiently on ARM platforms without dependency on Math Kernel Library (MKL). To achieve this, we utilized OpenBLAS BLAS library optimized for ARM and compatible with Scalable Vector Extension (SVE).

Figure 2 illustrates our contribution to enable oneDAL for ARM architecture.

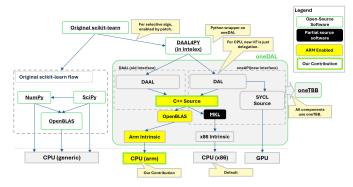


Fig. 2: Proposed Architecture: ARM SVE optimized oneDAL

The stages for ARM enablement and adoption of Open-BLAS as a computational engine include:

- Introduction of New Template: This research incorporated new template parameters and macros specific to ARM architectures as addition to existing architectures. Which allow more tailored and efficient execution of algorithms on ARM hardware without disrupting existing oneDAL functionality and architectures support.
- Refactoring Kernel Definitions and ARM-Specific Templates: Architecture-specific kernel definitions were refactored, and ARM-specific template parameters were introduced to improve code organization and maintainability. These changes allow oneDAL to run efficiently on both ARM and x86 architectures without compromising code integrity. The refactored structure makes it easier to distinguish between ARM and x86-specific implementations, while ensuring that algorithms are tailored for ARM hardware.
- Dynamic CPU Dispatch Mechanism: To handle diverse ARM configurations, we implemented a CPU dispatch mechanism that dynamically selects the most appropriate vectorized instructions (NEON or SVE) based on the ARM CPU's capabilities. This mechanism ensures that oneDAL utilizes the most efficient vectorization approach available on the target hardware.
- Conditional Compilation: ARM-specific optimizations
  were integrated using conditional compilation, which isolates ARM code paths from x86 paths. Compiler macros
  enable selective inclusion of ARM-specific code only
  when compiling for ARM.

```
#ifdef __ARM_SVE
// SVE-optimized code
#else
// Other_Architecture code
#endif
```

• Tailored Build System: The build system was adapted to detect the target architecture and apply appropriate compiler flags. ARM-specific flags activate SVE.

This comprehensive approach enabled the successful adaptation of *oneDAL* for ARM, ensuring cross-platform compatibility. With the core ARM enablement in place, we shifted our focus towards optimizing key computational routines and expanding the support for additional algorithms on the ARM platform.

# B. Sparse BLAS Implementations

Sparse matrix operations are essential in high-performance computing, particularly in machine learning (ML) and data analytics workloads, where large and sparse datasets are common. In *oneDAL*, three key routines are required to handle data in the Compressed Sparse Row (CSR) format: csrmm, csrmultd, and csrmv. Unlike BLAS and LAPACK standards, no universally accepted standard exists for sparse matrix operations [38]. Traditionally, *oneDAL* relies on MKL for these operations. However, it cannot be ported directly to ARM. To address this, we developed reference C++ imple-

mentations for these sparse routines based on MKL's functionality specifications, enabling compatibility with ARM and other platforms. This implementation is crucial for enabling algorithms such as PCA, covariance, correlation, and KMeans on ARM-based systems.

The sparse routines required by *oneDAL* are defined as follows:

- 1) **csrmm**:  $C \leftarrow \alpha op(A)B + \beta C$ , where A is a sparse matrix in CSR format and B, C are dense matrices.
- 2) **csrmultd**: C := op(A)B, where A and B are sparse matrices in the CSR format and C is a dense matrix.
- 3) **csrmv**:  $y \leftarrow \alpha op(A)x + \beta y$ , where A is a sparse matrix in the CSR format and x, y are densely formatted vectors.

Here op is either the identity or transpose operation.

The function *csrmm* already had a reference implementation. So, following will focus on the implementation details of the other two.

1) csrmultd: This routine requires A and B in 3-array CSR form, with 1-based indexing on the index arrays. Matrix C is dense and is stored in column-major format. The routine supports both AB and  $A^tB$ , therefore separate compute kernels for these operations were implemented.

For AB, the (i, j)th entry of C is obtained by iterating the below method over k.

$$C_{ij} \leftarrow A_{ik}B_{kj}$$

The entire matrix is obtained by iterating over i and j. The ideal order would be a row-wise iteration over A and B (because they are stored in CSR format); and a column-wise iteration over C (because it is stored in column major order). However, the column traversal of C and the row traversal of A cannot be achieved at the same time, since they share the same column index (i).

Therefore, this work has to choose between the following options:

- a). Row traversal on A and column traversal on C;
- b). Column traversal on A and row traversal on C.

CSR format makes the second option complicated, so this work decided to use the former. The nested loops are in the order j-k-i (innermost to outermost). For  $A^tB$ , the (i,j)th entry of C is obtained by iterating

$$C_{ij} \leftarrow A_{ki}B_{kj}$$

over i, j, k. Here, the ideal order of iterations, which is column-wise traversal over C and row-wise traversal over A and B, can be achieved by using i-j-k (inner to outer) as the order of iteration. This work implementation does the same.

2) csrmv: This routine requires A in 4-array CSR form. The index arrays can be either zero-based or one-based. Similarly to the previous case, separate kernels were implemented for Ax + y and  $A^tx + y$ . The choice of the order of iteration is obvious in this case because there are only two choices, of which this work chooses the one that involves a row-order traversal of A.

Initial performance evaluations indicate that while these implementations do not yet match the speed of MKL, they provide essential sparse matrix functionality for ARM architectures. These implementations establish a foundation for future optimizations, targeting ARM-specific architectural features to improve efficiency and close the performance gap with MKL.

# C. Vector Statistical Library (VSL) Implementations

The *oneDAL* library's Vector Statistical Library (VSL) traditionally relies on MKL for performing statistical computations, which has restricted its functionality on ARM-based platforms and led to compilation challenges for certain algorithms. To overcome these limitations, ARM-compatible implementations of essential VSL routines were developed. This section highlights the ARM-optimized implementations of two key routines: x2c\_mom, used for variance calculations, and xcp, for computing cross-products. These enhancements enable *oneDAL* to execute statistical operations independently of MKL, fully utilizing ARM's architecture.

1) Variance Calculation (x2c\_mom): The x2c\_mom kernel computes the sample variance across each coordinate for a dataset matrix  $X \in \mathbb{R}^{p \times n}$ , where each column represents a p-dimensional sample. The variance  $v_i$  for the i-th coordinate is defined as:

$$v_i = \frac{1}{n-1} \sum_{j=1}^{n} (X_{ij} - \mu_i)^2$$
 (1)

where  $\mu_i$ , the sample mean of the *i*-th coordinate, is calculated by:

$$\mu_i = \frac{1}{n} \sum_{i=1}^n X_{ij}.$$
 (2)

To enhance performance, this expression was reformulated using the first and second raw moments,  $S_i^{(1)}$  and  $S_i^{(2)}$ , as follows:

$$v_i = \frac{S_i^{(2)}}{n-1} - \frac{(S_i^{(1)})^2}{n(n-1)} \tag{3}$$

where:

$$S_i^{(1)} = \sum_{i=1}^n X_{ij}$$
 and  $S_i^{(2)} = \sum_{i=1}^n X_{ij}^2$ .

This formulation minimizes recalculations of means, thus facilitating vectorization and parallel execution using ARM SVE.

2) Matrix of Cross Products (xcp): The xcp kernel calculates the cross-product matrix  $C \in \mathbb{R}^{p \times p}$  for a dataset  $X \in \mathbb{R}^{p \times n}$ . Each element  $C_{ij}$  of this matrix is defined as:

$$C_{ij} = \sum_{k=1}^{n} (X_{ik} - \mu_i)(X_{jk} - \mu_j)$$
 (4)

where  $\mu_i$  and  $\mu_j$  are the means along coordinates i and j.

The routine supports batch-wise computation, where the cross-product matrix, the sum, and the number of observations from the previous batch are passed to the function. The function then updates the cross-product matrix to reflect

the combined data. For data processed in two batches, with  $1, \ldots, n'$  observations in the first batch and  $n'+1, \ldots, n'+n$  in the second, the cross-product matrix for each element  $C_{ij}$  is calculated as:

$$C_{ij} = \sum_{k=1}^{n'} \left( (X_{ik} - \mu'_i) + (\mu'_i - \mu_i) \right) \times \left( (X_{jk} - \mu'_j) + (\mu'_j - \mu_j) \right)$$

$$+ \sum_{k=n'+1}^{n'+n} (X_{ik} - \mu_i)(X_{jk} - \mu_j).$$
(5)

This expression is optimized by using:

$$C \leftarrow C' + \frac{S'(S')^T}{n'} - \frac{SS^T}{n} + XX^T, \tag{6}$$

where S' is the raw sum of the first batch, S the cumulative raw sum, and C' the previously computed cross-product matrix. Leveraging BLAS routines, this formula allows for memory-efficient computation, further improved by ARM SVE-based parallel processing.

This implementation delivers two new VSL implementation to open source community over traditional scalar approaches. These optimizations establish *oneDAL* as a viable library for data-intensive applications on ARM platforms.

#### D. Random Number Generation (RNG) Optimization

On ARM-based architectures, *oneDAL* faced limitations in random number generation (RNG) due to its reliance on the stdc++ implementation, which lacks advanced RNG capabilities critical for data science and machine learning workflows. To bridge this gap and achieve feature parity across platforms, we integrated OpenRNG as the RNG backend for *oneDAL* on ARM. OpenRNG is an open-source library offering a robust set of RNG functionalities, designed to closely replicate the interface and performance of MKL RNG, while incorporating optimized kernels tailored for high-performance computation.

1) Integration of OpenRNG into oneDAL: The integration of OpenRNG required several modifications to the oneDAL codebase to replace the default stdc++RNG backend. This involved creating a new header file, service\_rng\_openrng.h, which facilitates the interaction between oneDAL and OpenRNG. Since the OpenRNG interface is highly compatible with the MKL RNG interface, minimal changes were required in the existing RNG service files, specifically in service\_stat\_mkl.h.

Key modifications included:

- Developing a new service\_rng\_openrng.h header file to define the interface for OpenRNG.
- Adjusting service\_stat\_mkl.h to accommodate the interface similarities between MKL RNG and OpenRNG, thereby enabling seamless integration.
- Updating build configuration files to enable static linking with the OpenRNG library, ensuring that the advanced RNG functionalities are available in the final build across all supported hardware platforms.

- 2) Enhanced RNG Functionality and Performance Implications: The integration of OpenRNG into oneDAL introduces an expanded set of RNG capabilities on ARM platforms. Although RNG represents a relatively minor part of the overall computational workload, the integration of OpenRNG allows oneDAL to maintain functionality parity across platforms.
  - Support for Multiple RNG Engines: OpenRNG conforms to the MKL VSL RNG specification, supporting engines such as MT19937 and mcg59. Currently, mt2203 is not included in OpenRNG, but adding it could further improve performance for algorithms like Random Forests on ARM. In comparison, stdc++ only supports the MT19937 engine.
  - Optional Backend Configuration: Users can now select their preferred RNG backend at compile time, allowing flexibility between OpenRNG on ARM and MKL on x86, thereby promoting cross-platform consistency with tailored optimizations for each architecture.
  - Performance Parity and Cross-Platform Consistency: Benchmark results Figure 3 indicate that while RNG contributes minimally to the total workload time, OpenRNG on ARM provides comparable functionality to MKL on x86, achieving uniform RNG performance across both ARM and x86 systems.

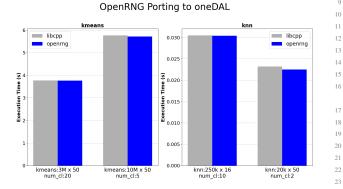


Fig. 3: Performance of KNN and KMeans with libcpp vs.  $^{24}_{25}$  OpenRNG

Figure 3 highlights the performance of RNG-dependent algorithms such as KNN and KMeans using the libcpp and OpenRNG backends, demonstrating that the adoption of OpenRNG maintains competitive performance without incurring additional overhead. By integrating OpenRNG, oneDAL gains a robust and consistent RNG backend for ARM architectures, enhancing its support for high-performance, ARM-optimized data analytics workflows while ensuring consistent performance and reproducibility across diverse hardware platforms.

# E. SVM Optimization for ARM optimized oneDAL Using SVE

Support Vector Machines (SVM) algorithms can be computationally intensive, especially with large datasets, due to the increased number of constraints that need to be optimized to

maximize the margin of separation. The oneDAL library provides an accelerated version of SVM, utilizing a Working Set Selection (WSS) mechanism based on second-order information to handle this complexity efficiently. Specifically, oneDAL uses the WSS3 variant, where pairs of indices are selected in each iteration for optimizing the Lagrange multipliers.

In this work, we focus on optimizing the 'WSSj' function in oneDAL's SVM implementation for ARM architectures with Scalable Vector Extension (SVE). The 'WSSj' function selects the index *j* to update the Lagrange multipliers. Although compilers typically vectorize loops, the data dependencies in 'WSSj' hinder automatic vectorization. ARM's SVE, with its predicate capabilities, offers an effective solution to this issue, enabling conditional operations within vectorized loops.

1) Original Loop in WSSj Function: The original scalar loop in the 'WSSj' function iterates over a range defined by  $j_{\text{start}}$  to  $j_{\text{end}}$ . Multiple if conditions filter values based on specific criteria, which complicates automatic vectorization. Below is the original loop implementation in C++.

```
for (size_t j = jStart; j < jEnd; j++)</pre>
    const algorithmFPType gradj = grad[j];
    if (!(I[j] & sign)) {
        continue;
    if ((I[j] & low) != low) {
        continue;
    if (gradj > GMax2) {
        GMax2 = gradj;
    if (gradj < GMin) {</pre>
        continue;
    const algorithmFPType b = GMin - gradj;
    algorithmFPType a = Kii + kernelDiag[j] - two *
    KiBlock[j - jStart];
    if (a <= zero) {</pre>
        a = tau;
    const algorithmFPType dt = b / a;
    const algorithmFPType objFunc = b * dt;
       (objFunc > GMax) {
        GMax = objFunc;
        Bj = j;
        delta = -dt;
}
```

Listing 1: Original Loop in WSS † Function

2) SVE-Optimized Loop in WSSj Function: To overcome the vectorization limitation, we implemented an SVE optimized version of the loop using predicates. ARM SVE predicates allow for conditionally executing operations within vectorized loops, thus handling the multiple if conditions in a single pass. The optimized loop leverages SVE's vector length adaptability, ensuring compatibility across ARM systems with different vector widths.

```
for (size_t j_cur = jStart; j_cur < jEnd; j_cur += w
) {
    svint32_t Bj_vec_cur = svindex_s32(j_cur, 1);
    // Vectorized index for Bj
    svbool_t pg2 = svwhilelt_b32(j_cur, jEnd);
    // Predicate for vector length adaptation</pre>
```

```
svint32_t vec_I = svld1sb_s32(pg2,
    reinterpret_cast<const int8_t *>(&I[j_cur])); //
    Load condition vector

// Combine the 'if' conditions
svint32_t result_of_and32 = svand_s32_m(pg2,
    vec_I, vecSignLow);
pg2 = svcmpeq_s32(pg2, result_of_and32,
    vecSignLow); // Predicate-based filtering
// Remaining vectorized logic...
// Remaining vectorized logic...
```

Listing 2: SVE-Optimized Loop in WSS j Function

In this SVE-optimized loop:

- **Predicate Initialization**: The variable pg2 adapts to the available vector length, allowing the code to run efficiently on ARM hardware with varying vector lengths.
- Combined Conditions: The if conditions ! (I[j] & sign) and (I[j] & low) != low are merged using predicates, enabling selective processing within the vectorized loop.
- Vectorized Execution: The loop executes vectorized operations across multiple indices simultaneously, dynamically adjusting to the hardware's vector length.
- *3) Performance Gains:* After implementing the SVE vectorized version of the WSSj function, we achieved substantial performance improvements shown in Figure 4.

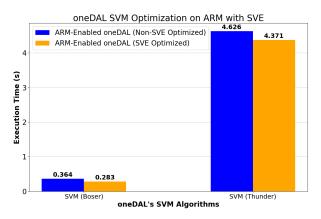


Fig. 4: Performance of SVM Non-SVE vs. SVE Optimized

- The **Boser method** of the SVM algorithm demonstrated a **22% performance gain**.
- The Thunder method exhibited a 5% performance gain.

These performance gains were achieved on AWS Graviton3 instances, which feature an SVE vector length of 256 bits, under a single-core configuration. The optimized vectorized loop maintained bitwise accuracy compared to its scalar base implementation, ensuring the fidelity of results. The successful vectorization of WSSj and the associated performance improvements highlight the potential of ARM SVE in enhancing SVM computations within oneDAL on ARM platforms.

## F. Experimental Setup

1) Hardware Specifications: Table I presents a comparison of the ARM-based (Graviton3) and x86-based (Ice-Lake) AWS instances used in this experiment to benchmark the performance of the ARM SVE optimized *oneDAL* in result section.

TABLE I: Configurations of ARM and x86 AWS Instances

	ARM Machine	x86 Machine
AWS Instance	c7g.8xlarge	c6i.8xlarge
vCPUs	32	32
Processor	AWS Graviton3	Intel Xeon 8375C
Clock Speed	2.5 GHz	3.5 GHz
Memory	32 GB	64 GB
Network Bandwidth	15 Gbps	12.5 Gbps
EBS Bandwidth	10 Gbps	10 Gbps
Price	\$0.7853/hr	\$1.36/hr

- 2) Development Environment: The development environment for the ARM SVE optimized *oneDAL* comprised:
  - Operating System: Ubuntu, running on the ARMv8 architecture.
  - Compiler and Libraries: GCC 13.0, LLVM 17.0, and OpenBLAS 0.3.26.
  - Profiling Tools: perf on Linux was used for profiling and performance analysis.

#### V. RESULTS

The ARM-SVE optimized *oneDAL* was assessed through an extensive series of benchmarks and practical applications to measure its performance on ARM-based systems. This section presents comparative benchmark results, showcasing the performance of ARM SVE optimized *oneDAL* against the original *scikit-learn* on ARM, as well as the x86 *oneDAL* with MKL backend. We utilize intel's scikit-learn benchmarks[39] to cover a range of machine learning models and datasets, including synthetic, real-world, and industry-relevant use cases. Additionally, results from the TPC-AI Benchmark [40], Data centric benchmarking i.e. DataPerf Selection Speech Challenge [41] hosted by Dataperf MLcommons [42], and a credit card fraud detection scenarios are included to highlight the practical value of our optimizations.

# A. Performance of ARM SVE Optimized oneDAL vs. Original scikit-learn on ARM Platform

The comparison between ARM SVE optimized *oneDAL* and the original *scikit-learn* demonstrates substantial performance gains on ARM architectures, as depicted in Figure 5. The performance improvements were observed across both training and inference tasks, with speedups ranging from 1x to as high as 217x. The most notable speedups were recorded for *SVM* on the *a9 dataset* (134.69x) and *SVM* on the *gisette dataset* (217.19x). These results confirm the significant impact of ARM SVE optimizations on the acceleration of machine learning algorithms.

However, not all algorithms benefited equally from the optimizations. For example, DBSCAN on the 500x3, 100

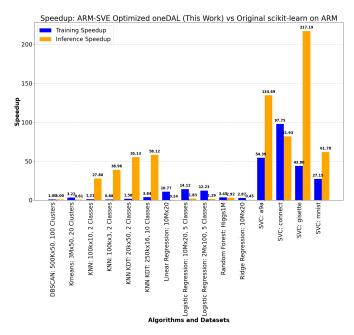


Fig. 5: Performance of ARM SVE optimized *oneDAL* vs. original *scikit-learn* 

clusters dataset showed no significant improvement, with a speedup of just 1.00x during training. This indicates that ARM SVE optimizations are less impactful for density-based clustering tasks, particularly when the dataset has small feature dimensions. Similarly, Logistic Regression on the 2Mx100, 5 classes dataset achieved a modest 1.29x speedup for inference, and linear models such as Linear Regression and Ridge Regression (both on the 10Mx20 dataset) demonstrated minimal improvements, with speedups of 0.24x (slower) and 0.45x (slower), respectively. These results suggest that further optimization is required in vectorized linear algebra operations to fully exploit the capabilities of ARM SVE. Although our optimizations lead to impressive performance improvements for most algorithms, there are specific cases, particularly with linear models and density-based clustering, where further refinement is needed to achieve maximum efficiency.

# B. Performance of ARM SVE Optimized oneDAL vs. x86 oneDAL (MKL)

The performance comparison of ARM SVE optimized *oneDAL* with x86 oneDAL with MKL backend reveals that the ARM-SVE optimizations deliver competitive speedups in both training and inference across a range of machine learning algorithms and datasets, as shown in Figure 5.

In the training phase, our work demonstrates upto 2.75x speedups over the default x86 oneDAL with MKL backend, with the highest improvements seen in KMeans (2.75x) and DBSCAN (1.92x) for clustering workloads. For KNN-based algorithms, our work achieves consistent speedups up to 1.5x, highlighting its effectiveness for distance-based operations. During inference, ARM-SVE optimized oneDAL maintains performance parity or achieves up to 1.83x speedup, partic-

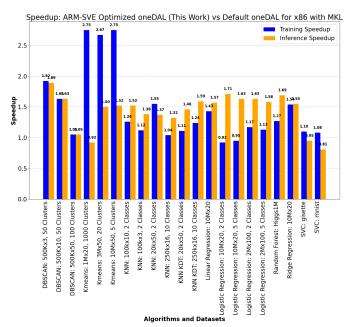


Fig. 6: Performance of ARM SVE optimized *oneDAL* vs. x86 oneDAL (MKL)

ularly excelling in workloads like DBSCAN, Logistic Regression and Linear Regression. Notably, algorithms with high computational complexity, such as SVM (SVC) and Random Forest, achieve comparable performance, further validating the optimization.

# C. Performance Analysis: DataPerf Selection Speech

The DataPerf Selection Speech benchmark evaluates dataset selection algorithms for keyword spotting, focusing on both training and inference execution times across three languages: English (en), Indonesian (id), and Portuguese (pt). The results are shown in Figure 7, which compares the performance of ARM-SVE optimized oneDAL, x86-based oneDAL with MKL, and original scikit-learn on ARM.

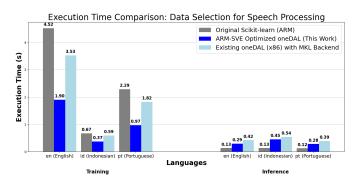


Fig. 7: Performance of DataPerf with ARM SVE optimized one DAL.

Our work exhibited significant improvements in training times compared to the original scikit-learn, with reductions of 58% (English), 45% (Indonesian), and 60% (Portuguese). This

was accompanied by modest gains over the implementation of oneDAL x86 (MKL), ranging from 37% to 46%. However, inference performance showed a mixed trend: while our version outperformed x86 oneDAL (MKL) across all languages and but still inference times were higher than original scikit-learn on ARM.

These results underscore the robustness of ARM-SVE optimizations oneDAL in accelerating data selection challenges, particularly in training, while narrowing the performance gap in inference.

#### D. Performance Analysis: TPC-AI Benchmark

The TPC-AI benchmark evaluates the end-to-end performance of machine learning workloads relevant to industry AI applications. We focused on the customer segmentation use case, which uses K-means clustering on a 1GB synthetic dataset [40].

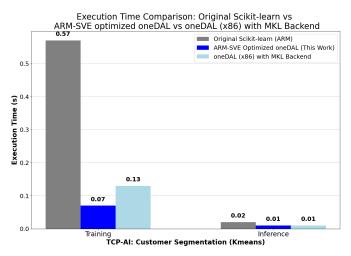


Fig. 8: Performance of TPC-AI Benchmark for ARM SVE optimized *oneDAL* 

As shown in Figure 8, the results for the customer segmentation task demonstrate significant performance improvements with ARM SVE optimized *oneDAL* compared to the original *scikit-learn* and x86 based *oneDAL* with MKL backend. Our work demonstrated significant improvements during training, achieving an approximate 87.72% reduction in execution time compared to the original *scikit-learn* and 46.15% compared to x86-based *oneDAL* with MKL backend. For inference, it achieved a 50% reduction in execution time relative to *scikit-learn*, performing equal to *oneDAL* on x86 with MKL. These results demonstrate that ARM-specific optimizations in *oneDAL* not only significantly improve training times but also ensure competitive performance for inference, matching or surpassing x86 MKL-based implementations in both aspects.

#### E. Real-World Use Case: Credit Card Fraud Detection

To demonstrate the impact of our work on the ARM ecosystem, we evaluated the ARM SVE optimized *oneDAL* on a credit card fraud detection dataset consisting of 284,807 transactions, including 492 fraud cases [43].

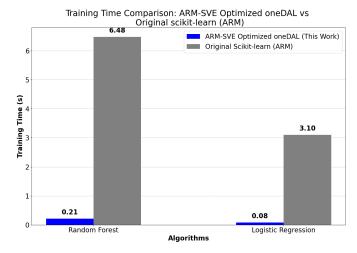


Fig. 9: Performance of Credit Card Fraud Detection with ARM SVE optimized *oneDAL*.

As illustrated in Figure 9, our work achieved substantial speedups over the original *scikit-learn* implementation on ARM SVE platform (Graviton3). It delivered a 31x speedup for random forest training and a 40x speedup for logistic regression compared to the original *scikit-learn*. These results demonstrate the significant acceleration provided by our optimizations, showcasing their impact on improving ML workloads in the ARM ecosystem.

## F. Future Directions

Future work could focus on optimizing oneDAL's underperforming machine learning models and also extending support to additional models to fully utilize ARM's SVE potential in ML applications. Collaborating with open-source initiatives, such as enhancing OpenRNG, VSL, and OpenBLAS with ARM-specific optimizations, could further improve the overall performance. Additionally, developing platform-agnostic optimization techniques within *oneDAL* to leverage SVE's vector-length flexibility would ensure consistent and robust performance across diverse ARM-based systems.

#### VI. CONCLUSION

The task of this research was to make valuable contributions to the open-source community by effectively optimizing UXL's oneAPI oneDAL for ARM Scalable Vector Extension (SVE) architectures, using the OpenBLAS library as a reference backend. The primary goal was to achieve performance parity with the conventional x86-based oneDAL architecture, which relies on MKL, to accelerate machine learning and data analytics on ARM platforms. To achieve this, the work involved identifying dependencies, mapping functions, refactoring code, modifying the build system, implementing sparse BLAS, and developing a novel VSL to ensure compatibility with ARM architecture. Extensive ARM-specific optimizations were carried out, including vectorization analysis and the use of ARM SVE intrinsics, particularly for Support Vector Machines (SVM), and were validated through rigorous testing

and validation processes. The study demonstrates that ARM architecture has the potential to provide high performance, contributing to the establishment of a more competitive and diversified HPC ecosystem.

The results demonstrate the effectiveness of the proposed approach, as validated by various benchmarks and real-world applications. The ARM-optimized *oneDAL* achieved comparable or superior performance relative to existing methods. Data science algorithms, including regression, classification, and clustering, showed significant improvements on standard benchmarks. Real-world applications, such as credit card fraud detection and customer segmentation, also experienced substantial performance enhancements.

## ACKNOWLEDGMENT

The authors thank the oneAPI and oneDAL development teams at the UXL Foundation for their ongoing assistance and for their thorough documentation that helped us comprehend the nuances of the library. We also thank the OpenBLAS, ARM, and Intel communities for their invaluable contributions and for upholding a robust open-source, high-performance library that allowed us to perform this work advancements.

In addition, the authors are grateful for the supportive input and ideas of colleagues and collaborators who came in at each stage of the study. We also thank the HPC community for creating a collaborative atmosphere that leads to innovative thinking. Such interactions have enriched this research through learning as well as sharing thoughts with like-minded peers.

#### REFERENCES

- J. Dongarra et al., "Enabling HPC for big data and extreme-scale computing: Software and hardware challenges," Int. J. High Performance Computing Appl., vol. 27, no. 2, pp. 172-182, 2013.
- [2] UXL Foundation, "oneDAL: oneAPI Data Analytics Library," [Online]. Available: https://github.com/oneapi-src/oneDAL. Accessed: Dec 01, 2024.
- [3] Grand View Research "High Performance Computing Market Size, Share and Trends Analysis Report" [Online]. Available: https://www.grandviewresearch.com/industry-analysis/ high-performance-computing-market. Accessed: Dec 01, 2024.
- [4] S. Matsuoka, "Fugaku and A64FX: the First Exascale Supercomputer and its Innovative Arm CPU," in 2021 Symposium on VLSI Circuits, Kyoto, Japan, 2021, pp. 1-3, doi: 10.23919/VLSICircuits52068.2021.9492415.
- [5] S. Fujitsu, "Supercomputer Fugaku," [Online]. Available: https://www. fujitsu.com/global/about/innovation/fugaku/. Accessed: Dec 01, 2024.
- [6] N. Stephens et al., "The ARM scalable vector extension," IEEE Micro, vol. 37, no. 2, pp. 26-39, 2017.
- [7] Wikipedia, "List of ARM processors," [Online]. Available: https://en. wikipedia.org/wiki/List\_of\_ARM\_processors. Accessed: Dec 01, 2024.
- [8] Fujitsu, "Next Arm Processor FUJITSU-MONAKA and Its Technologies," [Online]. Available: https://www.fujitsu.com/global/products/computing/servers/supercomputer/topics/sc23/. Accessed: Dec 01, 2024.
- [9] P. Sharma, "High Performance and Energy Efficient Processor for Next Generation Data Centres: FUJITSU - MONAKA," in 2023 IEEE 30th Int. Conf. on High Performance Computing, Data, and Analytics (HiPC), Goa, India, 2023, pp. xxiii-xxiii, doi: 10.1109/HiPC58850.2023.00012.
- [10] UXL Foundation, "Democratizing the use of AI: FUJITSU-MONAKA," [Online]. Available: https://www.oneapi.io/event-sessions/democratizing-the-use-of-ai-fujitsu-monaka-us-q423/. Accessed: Dec 02, 2024.
- [11] oneDAL [Online]. Available: https://oneapi-spec.uxlfoundation.org/ specifications/oneapi/latest/elements/onedal/source/ Accessed: Dec 02, 2024

- [12] Unified Acceleration Foundation (UXL), "The Unified Acceleration Foundation is an evolution of the oneAPI initiative," [Online]. Available: https://uxlfoundation.org/. Accessed: Dec 02, 2024.
- [13] Intel Corporation, "Fast, Scalable and Easy Machine Learning With DAAL4PY," [Online]. Available: https://intelpython.github.io/daal4py/. Accessed: Dec 02, 2024.
- [14] F. Pedregosa et al., "Scikit-learn: Machine Learning in Python," J. Mach. Learn. Res., vol. 12, pp. 2825–2830, 2011.
- [15] Intel Corporation, "Intel® oneAPI Math Kernel Library (MKL)," [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html. Accessed: Dec 02, 2024.
- [16] UXL Foundation, "System Requirements," [Online]. Available: https://uxlfoundation.github.io/oneDAL/system-requirements.html. Accessed: Dec 02, 2024.
- [17] Intel Corporation, "Intel® Advanced Vector Extensions 512", [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html. Accessed: Dec 02, 2024.
- [18] Neon [Online]. Available: https://developer.arm.com/Architectures/ Neon. Accessed: Dec 02, 2024.
- [19] Arm Developer, "Scalable Vector Extension (SVE)," [Online].

  Available: https://developer.arm.com/Architectures/Scalable%
  20Vector%20Extensions. Accessed: Dec 02, 2024.
- [20] OpenBLAS Developers, "OpenBLAS: An Optimized BLAS Library," [Online]. Available: https://www.openblas.net. Accessed: Dec 02, 2024.
- [21] S. Fujitsu, "Developer Story: How We Ported oneDNN to Fugaku with Arm," [Online]. Available: https://www.oneapi.io/blog/developer-story-how-we-ported-onednn-to-fugaku/. Accessed: Dec 02, 2024
- [22] C. R. Harris et al., "Array programming with NumPy," Nature, vol. 585, pp. 357–362, 2020, doi: 10.1038/s41586-020-2649-2.
- [23] P. Virtanen et al., "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," Nature Methods, vol. 17, pp. 261–272, 2020, doi: 10.1038/s41592-019-0686-2.
- [24] UXL Foundation, "Intelex Extension for Scikit-learn," [Online]. Available: https://uxlfoundation.github.io/scikit-learn-intelex/latest/index.html. Accessed: Dec 02, 2024.
- [25] Khronos Group "SYCL" [Online]. Available: https://en.wikipedia.org/ wiki/SYCL. Accessed: Dec 02, 2024.
- [26] Intel Corporation "Data Parallel C++: the oneAPI Implementation of SYCL\*" [Online]. Available: https://www.intel.com/content/www/us/ en/developer/tools/oneapi/data-parallel-c-plus-plus.html. Accessed: Dec 02, 2024.
- [27] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs," in *Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*, 2013.
- [28] X. Zhang , Q. Wang, and Y. Zhang, "Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor," in *IEEE 18th Int'l Conf. on Parallel and Distributed Systems (ICPADS)*, 2012.
- [29] OpenBLAS Developers, "GotoBLAS," [Online]. Available: https://en. wikipedia.org/wiki/GotoBLAS. Accessed: Dec 02, 2024.
- [30] OpenBLAS Developers, "OpenBLAS," [Online]. Available: https://en. wikipedia.org/wiki/OpenBLAS. Accessed: Dec 02, 2024.
- [31] Intel Corporation, "Intel® Performance Libraries (MKL, TBB, IPP, DAAL)," [Online]. Available: https://doku.lrz.de/ intel-performance-libraries-mkl-tbb-ipp-daal-11481683.html. Accessed: Dec 02, 2024.
- [32] C. Li, H. Jia, H. Cao, J. Yao, B. Shi, C. Xiang, J. Sun, P. Lu, and Y. Zhang, "AutoTSMM: An Auto-tuning Framework for Building High-Performance Tall-and-Skinny Matrix-Matrix Multiplication on CPUs," in *IEEE Int'l Symposium on Parallel and Distributed Processing with Applications*, 2021.
- [33] Intel, "Sparse Blas Routine," [Online]. Available: https://www.intel. com/content/www/us/en/docs/onemkl/developer-reference-c/2025-0/ overview.html. Accessed: Dec 02, 2024.
- [34] Intel, "Statistical Functions," [Online]. Available: https://www.intel. com/content/www/us/en/docs/onemkl/developer-reference-c/2025-0/ overview.html. Accessed: Dec 02, 2024.
- [35] ARM, "Arm Performance Libraries," [Online]. Available: https://developer.arm.com/documentation/101004/latest/. Accessed: Dec 02, 2024.
- [36] ARM, "OpenRNG: New Random Number Generator Library for best performance when porting to Arm," [Online].

- Available: https://community.arm.com/arm-community-blogs/b/tools-software-ides-blog/posts/openrng-24-04. Accessed: Dec 02, 2024
- [37] oneDAL (this work), "Enable ARM(SVE) CPU support with reference backend," [Online]. Available: https://github.com/oneapi-src/oneDAL/ null/2614. Accessed: Dec 02, 2024.
- pull/2614. Accessed: Dec 02, 2024.
  [38] OpenBLAS Developers, "OpenBLAS Faq," [Online]. Available: https://github.com/OpenMathLib/OpenBLAS/wiki/Faq. Accessed: Dec 02, 2024.
- [39] Intel Corporation, "scikit learn bench," [Online]. Available: https://github.com/IntelPython/scikit-learn\_bench. Accessed: Dec 02, 2024.
- [40] C. Brücke, P. Härtling, R. D. E. Palacios, H. Patel, and T. Rabl, "TPCx-AI An Industry Standard Benchmark for Artificial Intelligence and Machine Learning Systems," *Proc. VLDB Endow.*, vol. 16, no. 12, pp. 3649–3661, 2023, doi: 10.14778/3611540.3611554.
- [41] M. Mazumder et al., "DataPerf: Benchmarks for Data-Centric AI Development," in Neural Information Processing Systems Datasets and Benchmarks Track, 2023, [Online]. Available: https://openreview.net/ forum?id=LaFKTgrZMG.
- [42] MLCommons, "MLCommons Multilingual Spoken Words Dataset," [Online]. Available: https://mlcommons.org/datasets/ multilingual-spoken-words/. Accessed: Dec 02, 2024.
- [43] Kaggle, "Credit Card Fraud Detection," [Online]. Available: https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud. Accessed: Dec 02, 2024