Harnessing Arm Scalable Vector Extension (SVE) for Accelerated OpenBLAS and NumPy Performance

Presented By: Aniket P. Garade, Kuldeep Pal

Centre for Development of Advanced Computing, Bengaluru, India

Outline

- Motivation
- Vectorization
- SVE in General
- OpenBLAS
  - ➢ Our Implementation of Level-1 & Level-2 BLAS Operations
  - ➢ Performance Analysis of GEMV, SWAP, SCAL and ROT BLAS Routines
- NumPy
  - ➢ SVE Implementation and Performance Analysis of Basic Arithmetic, Dot Products, and GEMM on Arm Architecture
- Conclusion

Motivation

- Mathematical libraries are essential for high-performance computing, offering routines for BLAS and other core numerical operations..
- Effective optimization leverages advanced processors (ARM) to maximize performance.
- SIMD vectorization techniques enhance performance by reducing instruction counts and improving throughput.

Vectorization

Vectorization on ARM

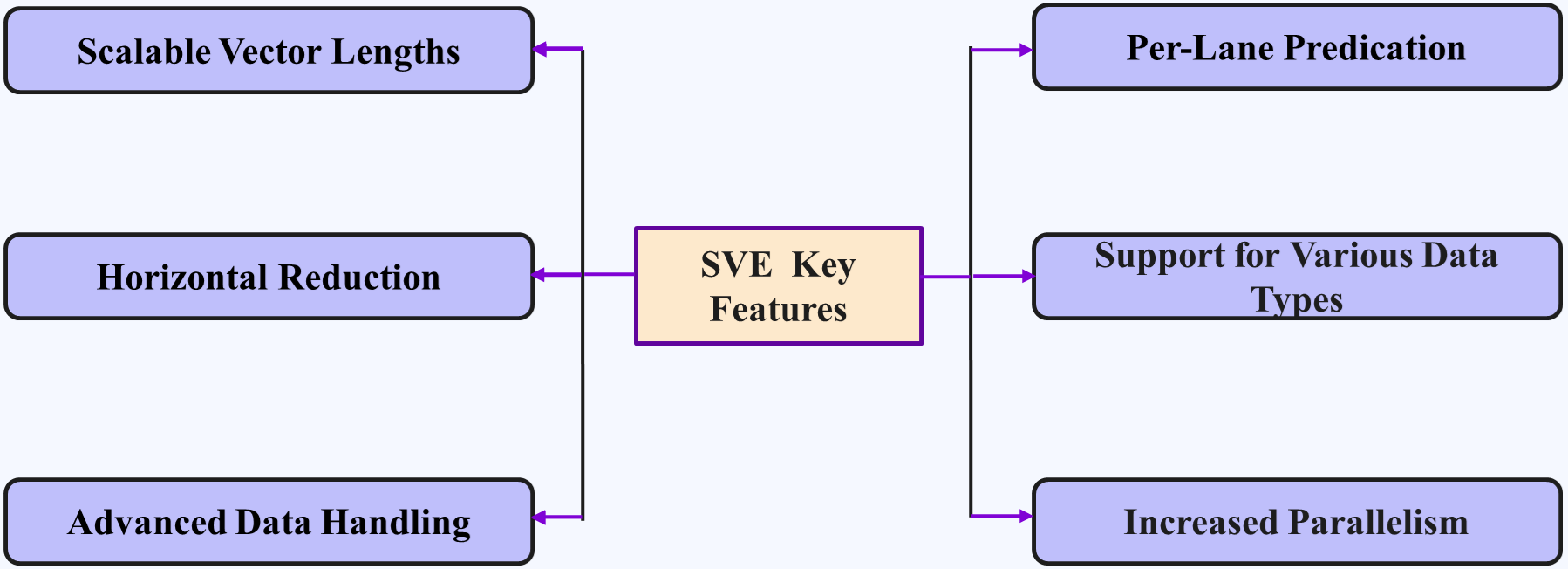
Neon:

- Fixed 128-bit SIMD for parallel data processing.
- Supports 8, 16, 32, 64-bit integers and 32, 64-bit floating-point numbers.
- Intended to improve audio, video encoding and decoding, 3D graphics , gaming and signal processing.

SVE:

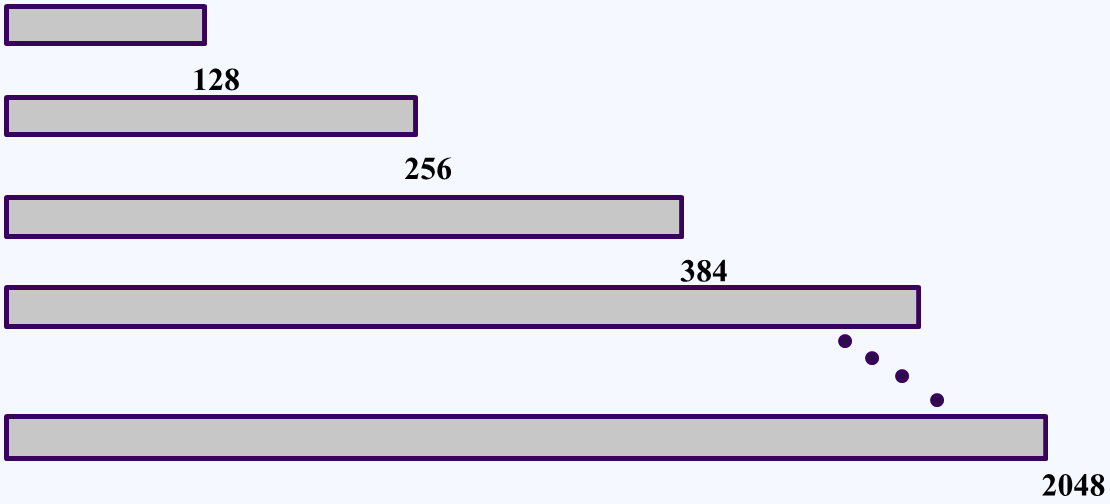
- Scalable SIMD with configurable vector length.
- Supports 8, 16, 32, 64-bit integers and 32, 64-bit floating-point numbers.
- Developed for HPC and AI applications

SVE Key Features



Scalable Vectors

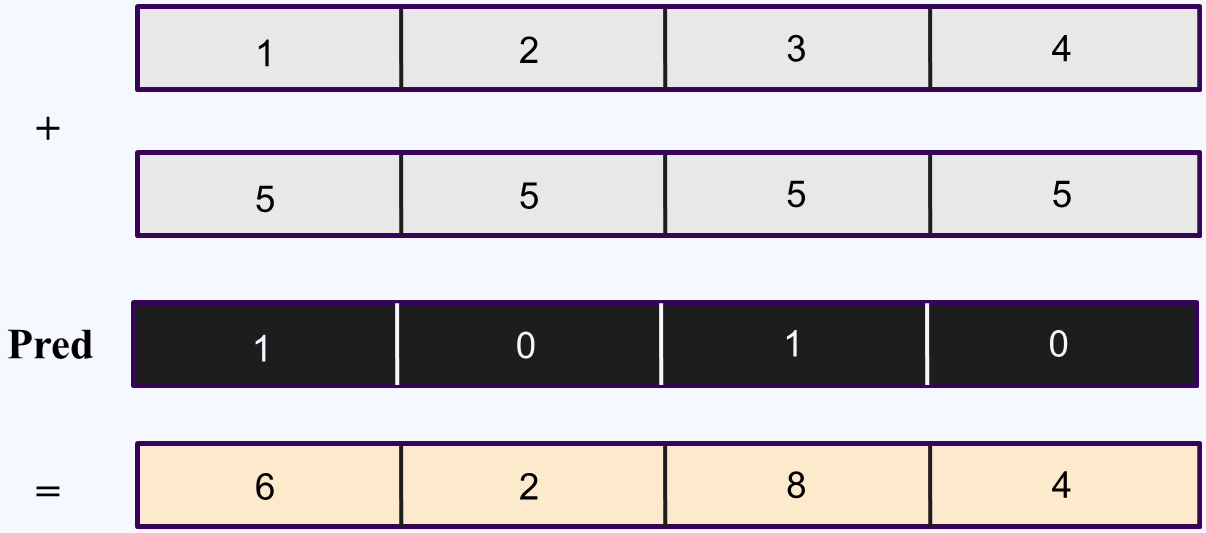
- Design to support variable vector lengths
- 128 bits to 2048 bits, in 128-bit increments



- **longer vectors, maximum parallelism**
- **Same code can run on different SVE implementation without any modification**

Per-Lane Predication

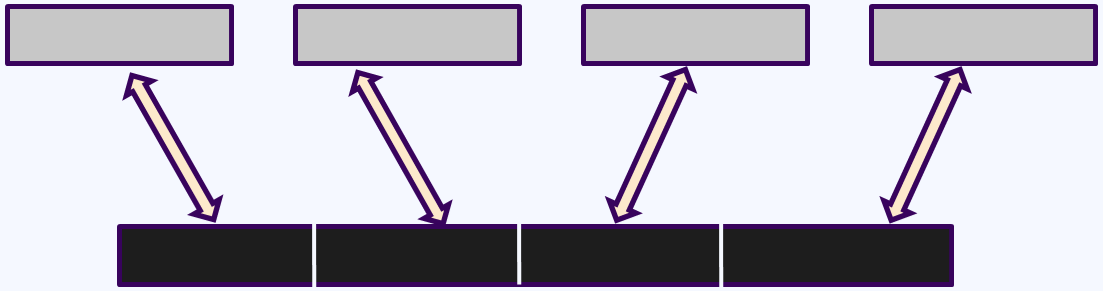
- Operation works on individual lanes under the control of Predicate Register



Gather Load and Scatter Store

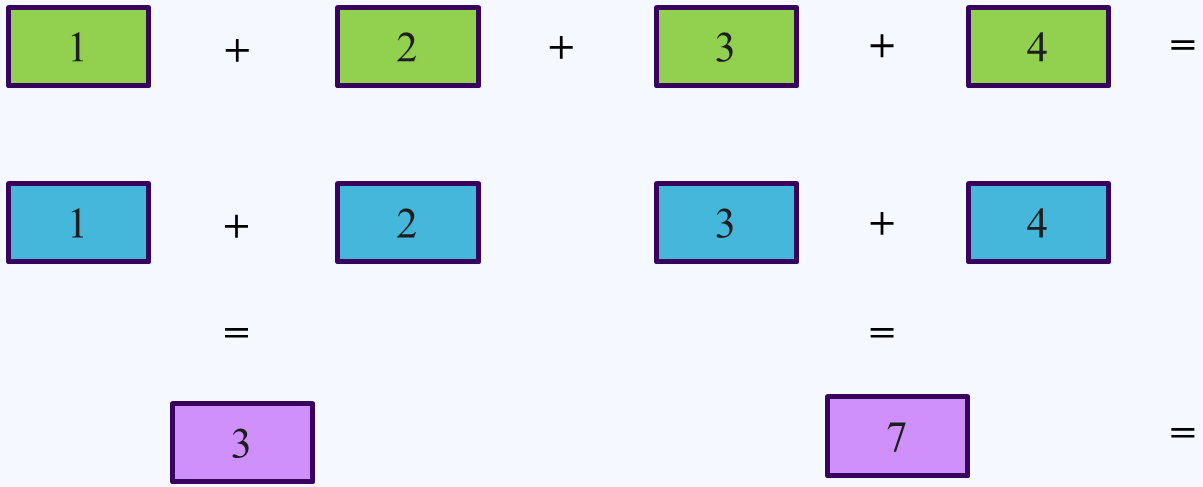
- **Gather Load:** Loads the data from the non-contiguous memory Locations into a vector register using index vector.

- **Scatter Store:** Stores data from a vector register back to non-contiguous locations.



Extended Floating point Horizontal Reduction

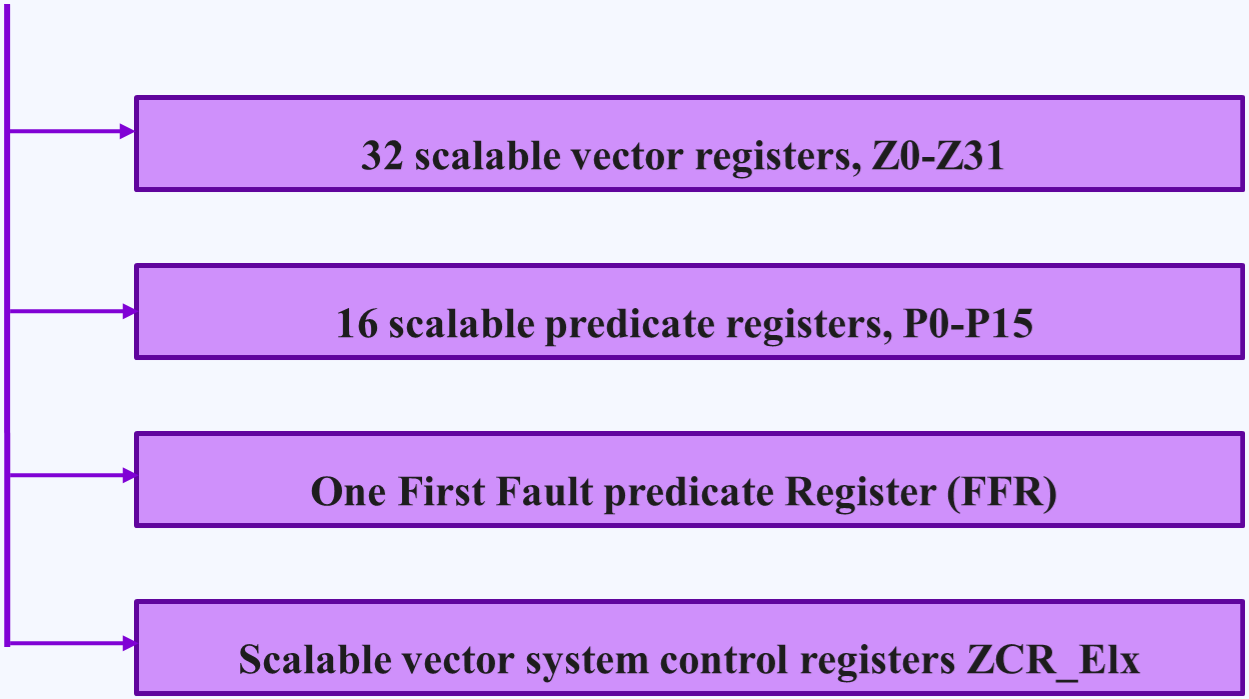
- Horizontal reduction reduces the elements of single vector.
- produce a single result or a smaller set of results



Data Type Support and Parallelism

- Supports 8, 16, 32, 64-bit integers and 32, 64-bit floating-point numbers.
- Leverages multicore systems for high-performance parallel computation.

SVE architecture fundamentals



Important SVE Intrinsics

Operation	SVE Intrinsics
Load/Store	**svld1 , svst1**
Arithmetic	**svadd** , **svsub**, **svmul** , svsqrt , svdiv,
Compare and Predicate Generation	svcmpeq, svcmpgt, svcmpge, svcmpne, **svwhilelt**
Logical	svand, svorr, sveor, svnot, svlsr, svlsl
Data Rearrangement	svtrn1, svtrn2, svrev
Reduction	**svaddv**, svmaxv, svminv
Duplication and Selection	**svdup**, svsel, svext, svins, svabs
Vector Length Management	svcntb, svcnth, **svcntw, svcntd**

SVE Sample Code (Dot Product Calculation)

```
float dotProduct(int n, float *x, float *y) {
    float result = 0.0;
    for (int i = 0; i < n; i++) {
        result += x[i] * y[i];
    }
    return result;
}
```

C Code

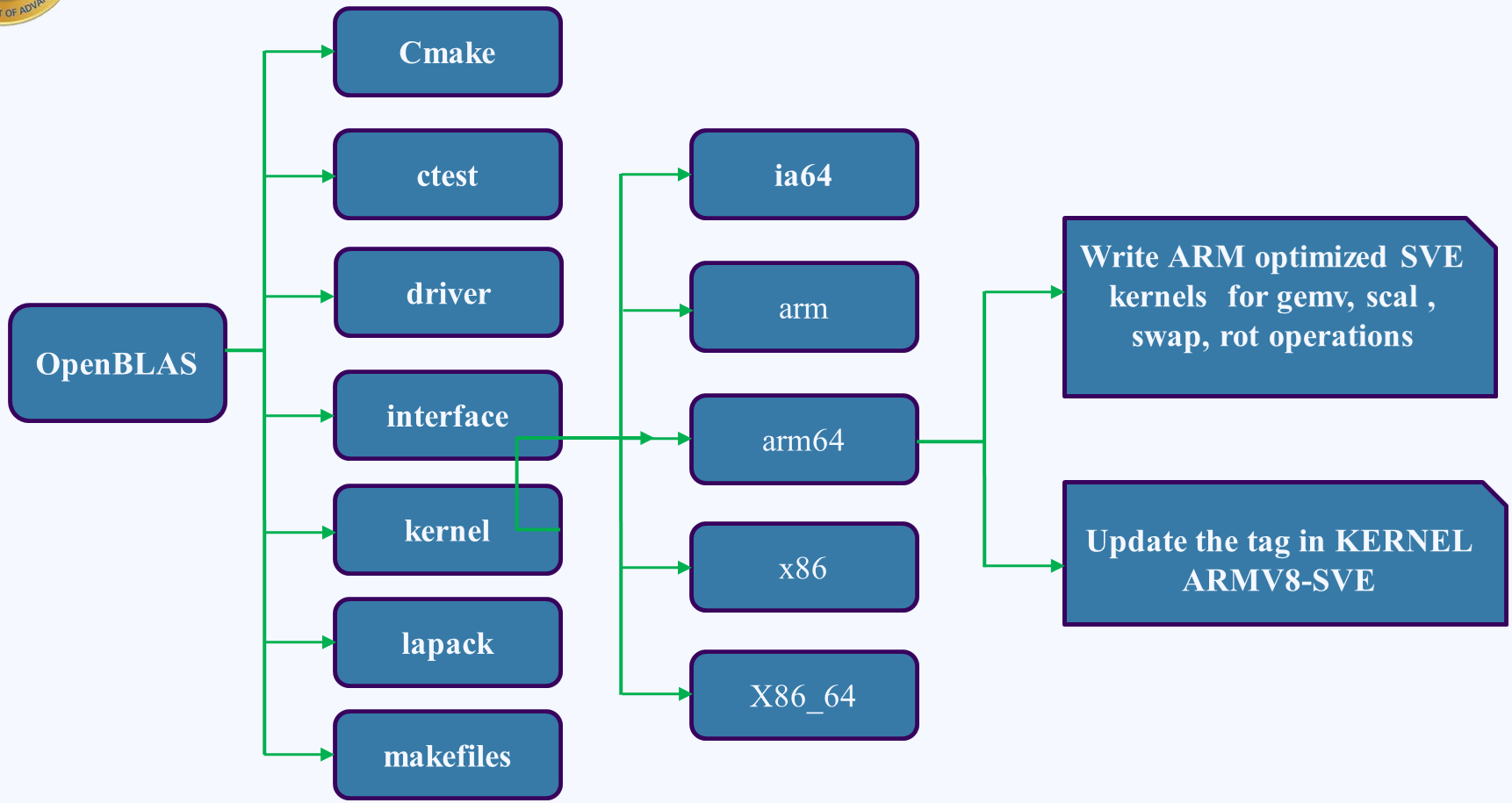
```
float dotProductUsingSVE(int  n, float *x, float *y)
{

    int sve_width = svcntw();
    svfloat32_t acc_a=svdup_f32(0.0);
    float result=(float)0;

        for (int i = 0; i < n; i += sve_width) {
        svbool_t pg = svwhilelt_b32(i, n);
        svfloat32_t  x_vec = svld1(pg, &x[i]);
        svfloat32_t  y_vec = svld1(pg, &y[i]);
         acc_a    =    svmla_m(pg,    acc_a,    x_vec,
y_vec);
    }          //          acc_a = acc_ + x_vec * y_vec

    return svaddv(svptrue_b32(), acc_a);
}
```

SVE Code

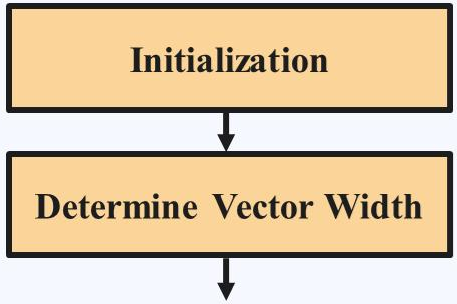

भारत संगणन विकास केंद्र
सी डैक
CDAC
॥ज्ञानादेव तु कैवल्यम्॥
CENTRE FOR DEVELOPMENT OF ADVANCED COMPUTING

OpenBLAS

OpenBLAS flow for SVE Optimization



Our SVE Implementation for L1 and L2 BLAS Operations



1. **Initialization:**

   Set up the necessary parameters, such as the **number of elements and scalar values**, and initialize accumulators or variables to zero.

2. **Determine Vector Width:**

   Determine the SVE vector width by using **svcntw()** or **svcntd()** SVE intrinsics.

A. P. Garade *et al.*, "Optimization Strategies to Accelerate BLAS Operations with ARM SVE," *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, Wakefield, MA, USA, 2024, pp. 1-7, doi: 10.1109/HPEC62836.2024.10938457
https://ieeexplore.ieee.org/document/10938457

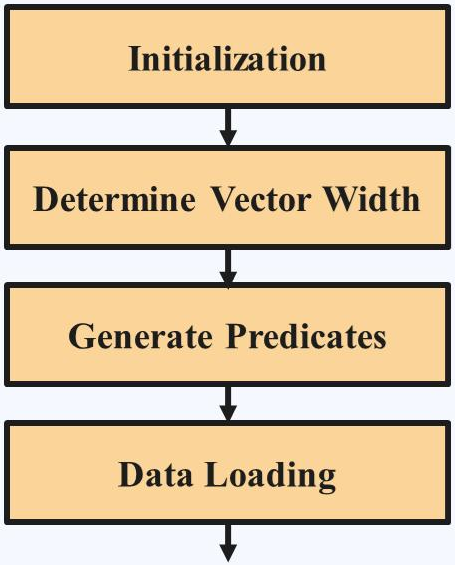
Proposed SVE Algorithm for L1 and L2 BLAS Operations

3. **Generate Predicates:**

   Generate predicates using **svwhilelt_b32 or svwhilelt_b64** to create masks that identify active data elements within each chunk**.**

4. **Data Loading:**

   Load data into SVE vector registers using **svld** intrinsics for loading of matrix segments or vectors.

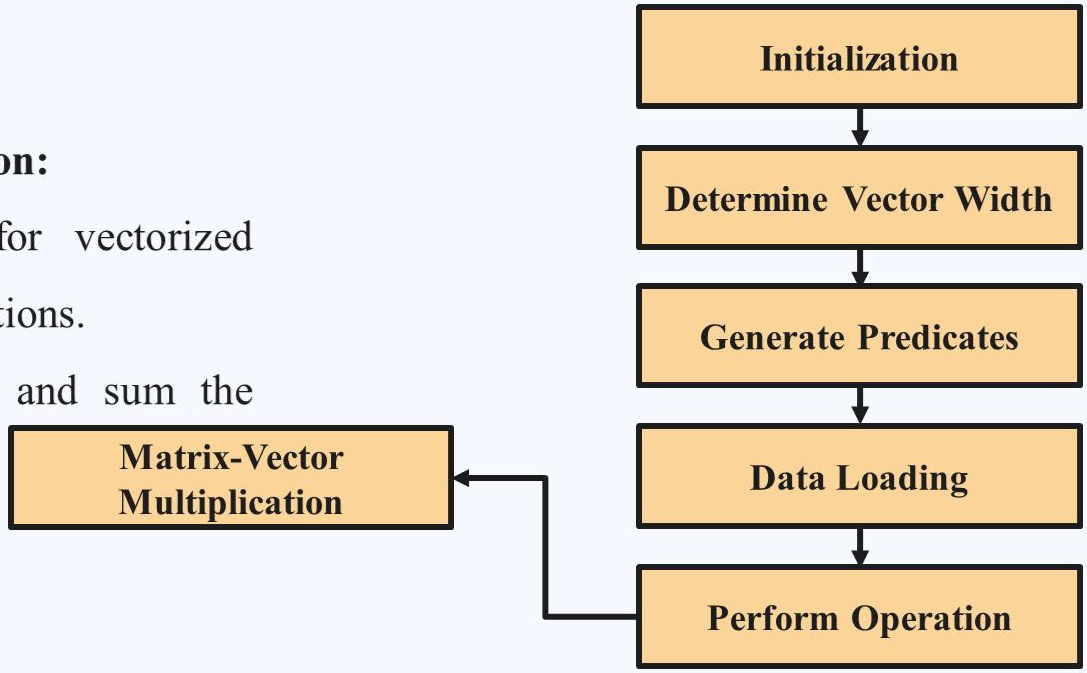


Proposed SVE Algorithm for L1 and L2 BLAS Operations

5. Perform Operations:

a) Matrix-Vector Multiplication:

- Use **svmla** intrinsics for vectorized multiply-accumulate operations.
- Apply **svaddv** to reduce and sum the accumulated results.

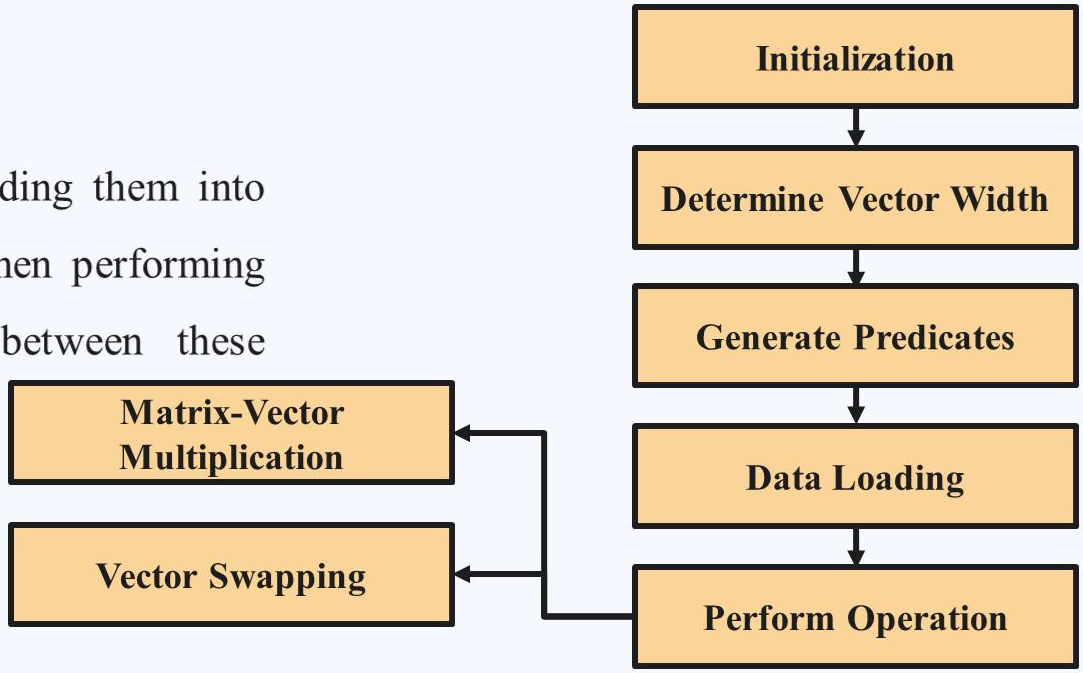


Proposed SVE Algorithm for L1 and L2 BLAS Operations

5. Perform Operations:

b) Vector Swapping:

Swap elements by first loading them into temporary SVE registers, then performing the exchange of values between these registers.

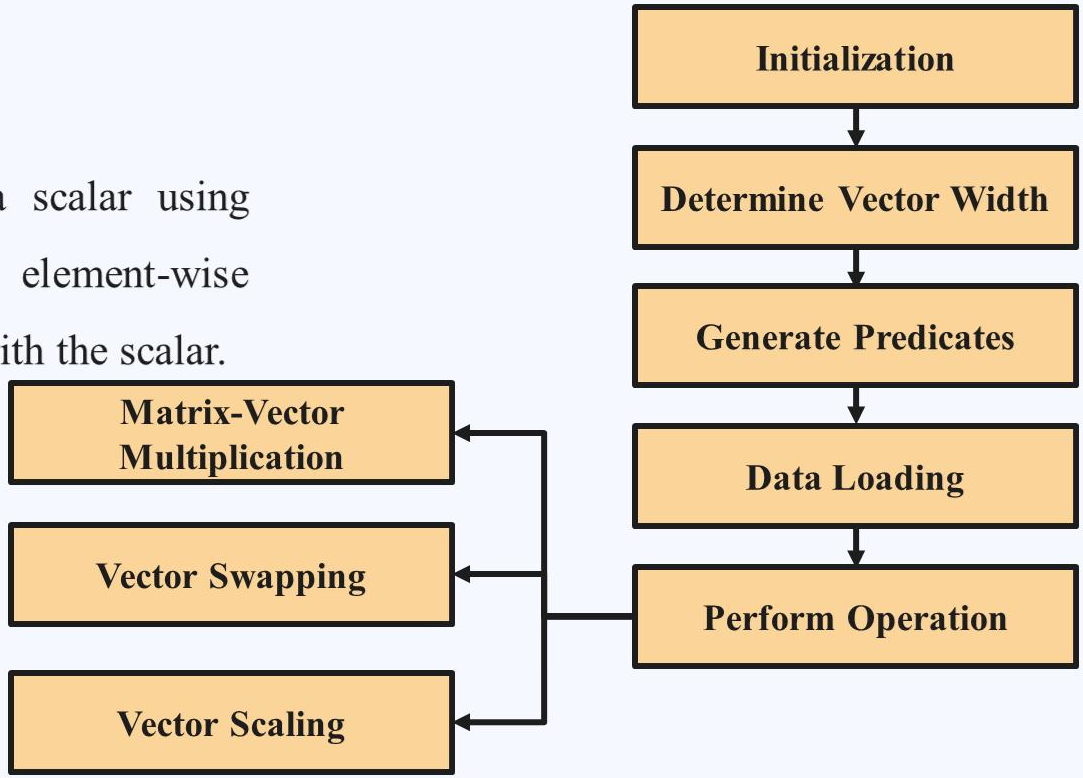


Proposed SVE Algorithm for L1 and L2 BLAS Operations

5. Perform Operations:

c) Vector Scaling:

Scale vector elements by a scalar using **svmul**, which performs element-wise multiplication of the vector with the scalar.



Proposed SVE Algorithm for L1 and L2 BLAS Operations

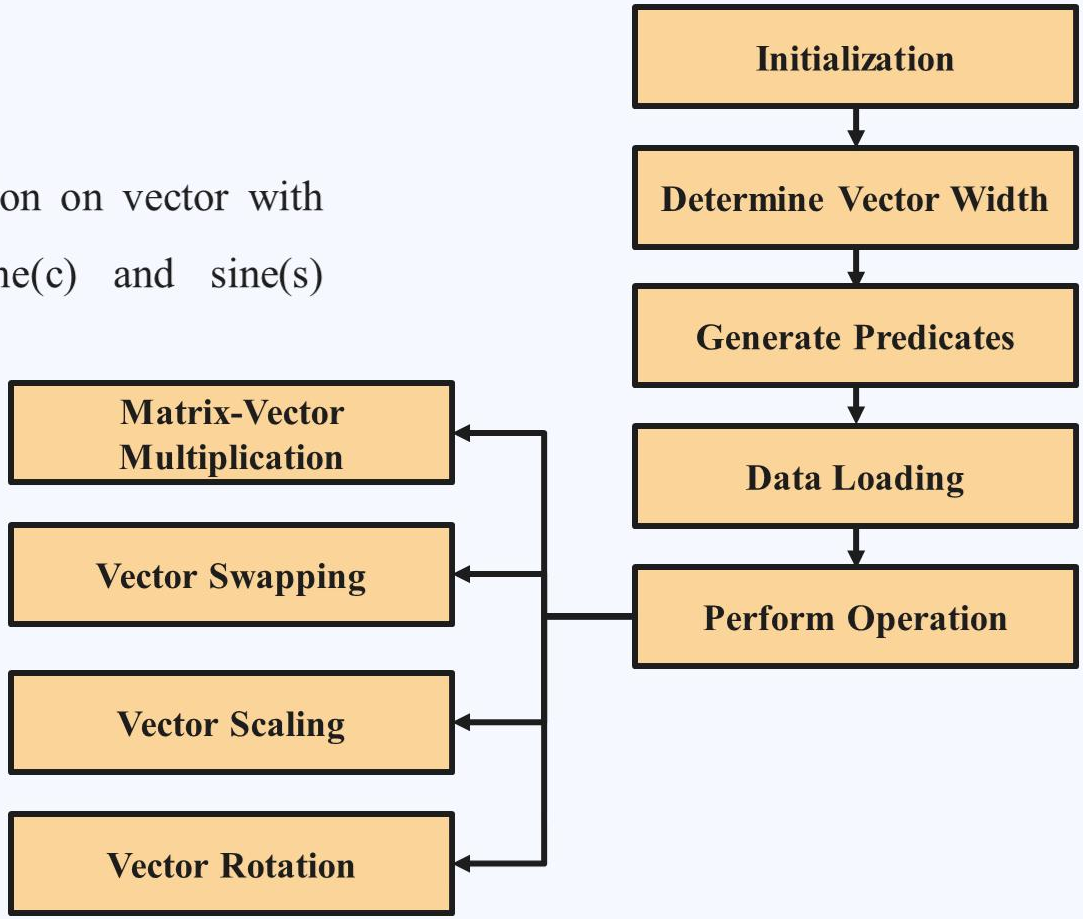
5. Perform Operations:

d) Vector Rotation:

- The given rotation operation on vector with rotation parameters cosine(c) and sine(s) expressed as follows:

$$xi' = c \cdot xi + s \cdot yi \quad (1)$$

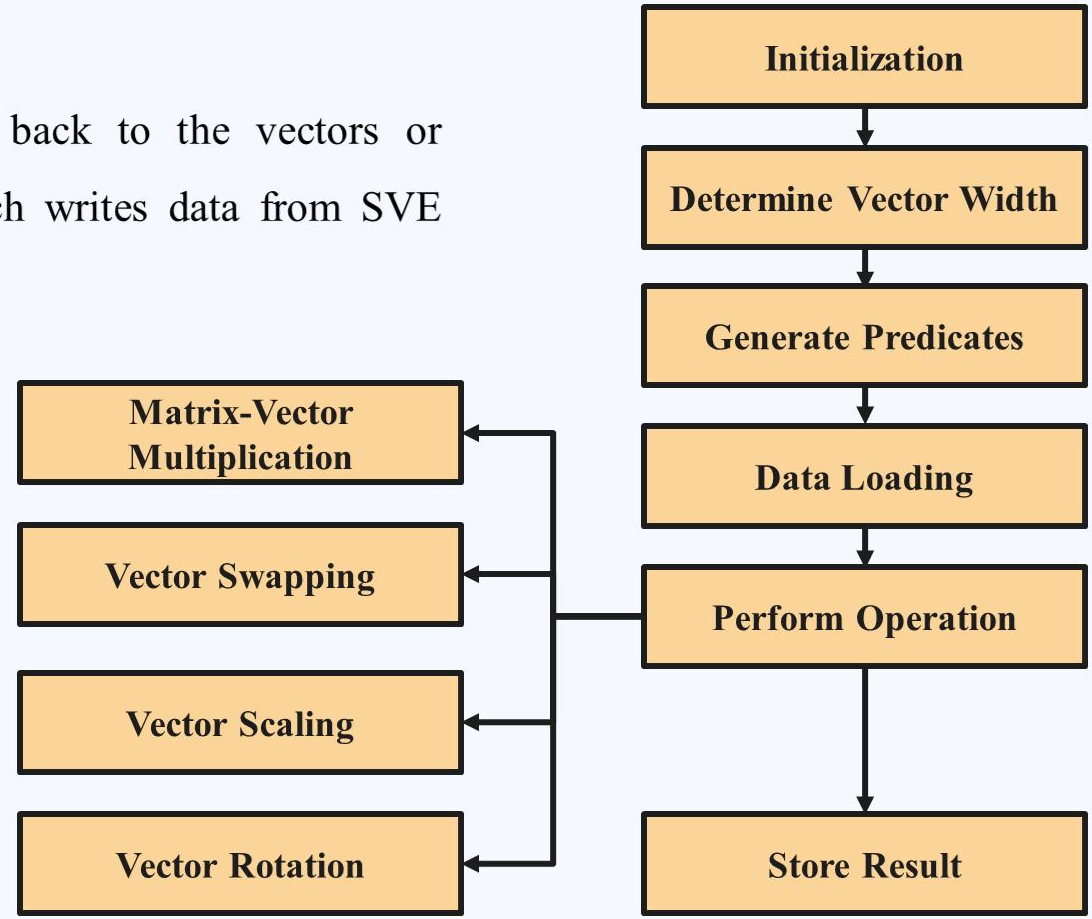
$$yi' = c \cdot yi - s \cdot xi \quad (2)$$



Proposed SVE Algorithm for L1 and L2 BLAS Operations

6. **Store Result:**

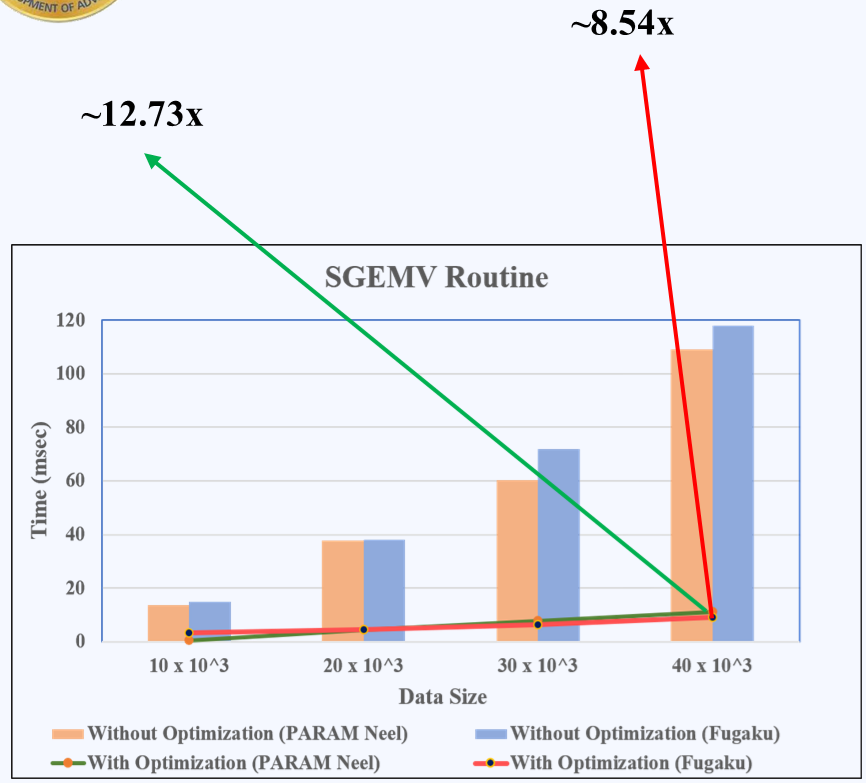
   Store the processed results back to the vectors or matrices by using **svst,** which writes data from SVE vector registers to memory**.**

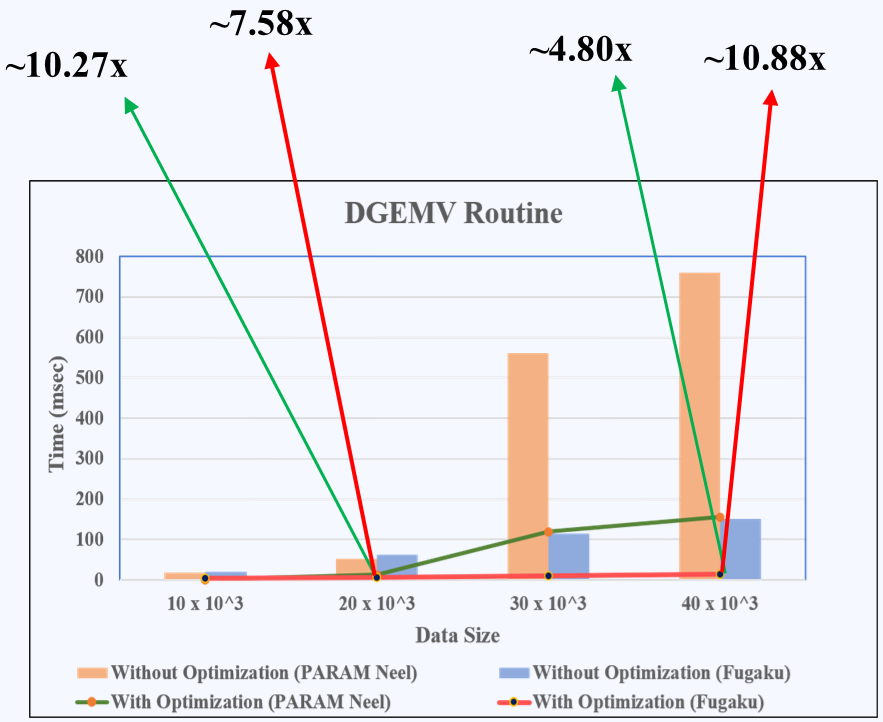


Experimental Setup for OpenBLAS SVE Optimization

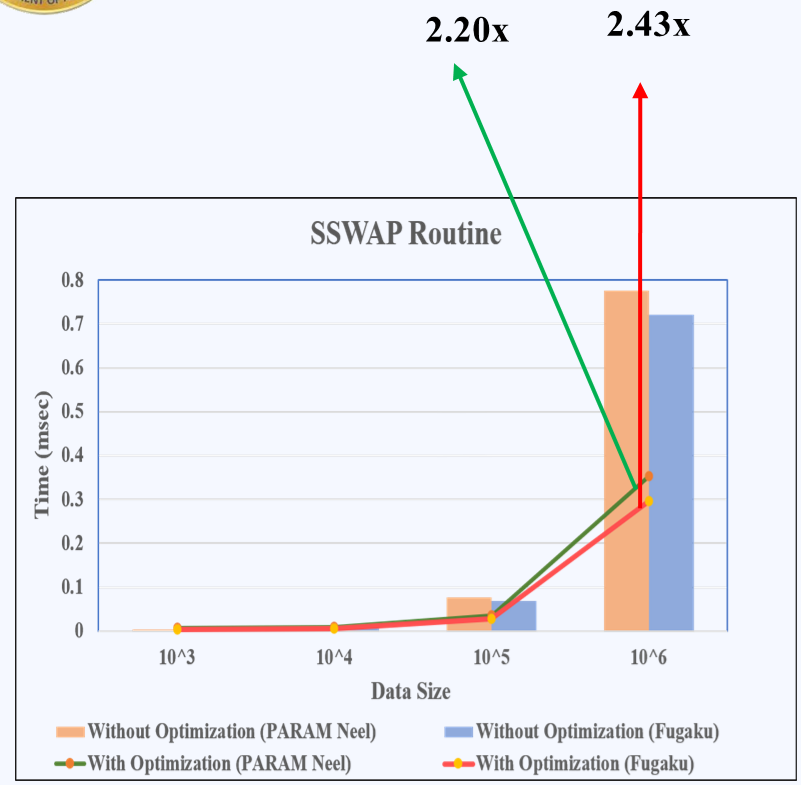
Info	PARAM Neel	Fugaku
Clock speed	1.8GHz	2.2GHz
Cores	48	52
Vectorization	SVE	SVE
Memory	32 GB HBM2	32 GB HBM2
Compiler	GCC-12.2	GCC-12.2

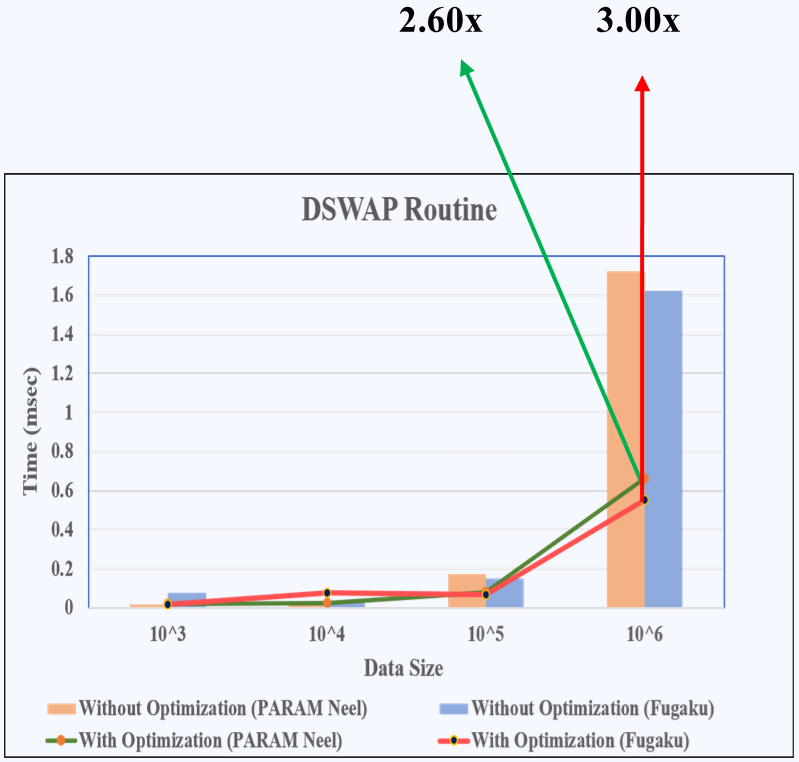
Performance Analysis of GEMV Routines



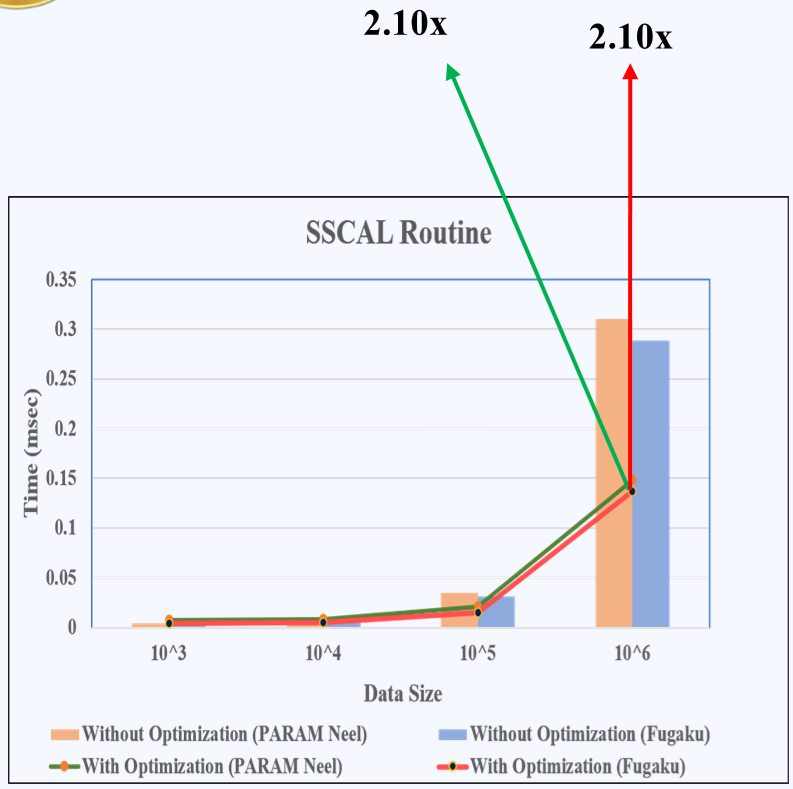


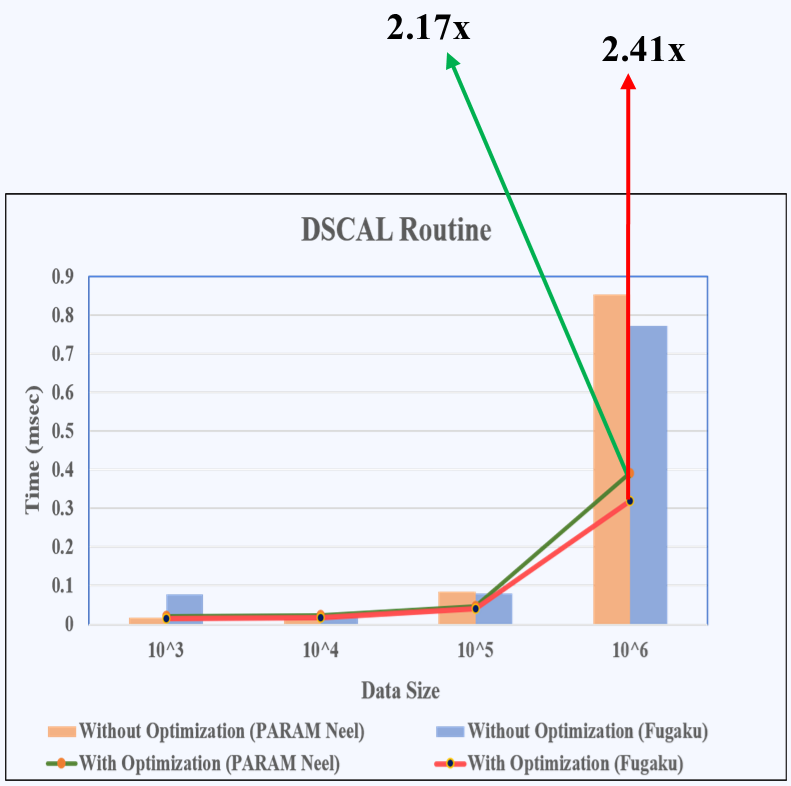
Performance Analysis Of SWAP Routine



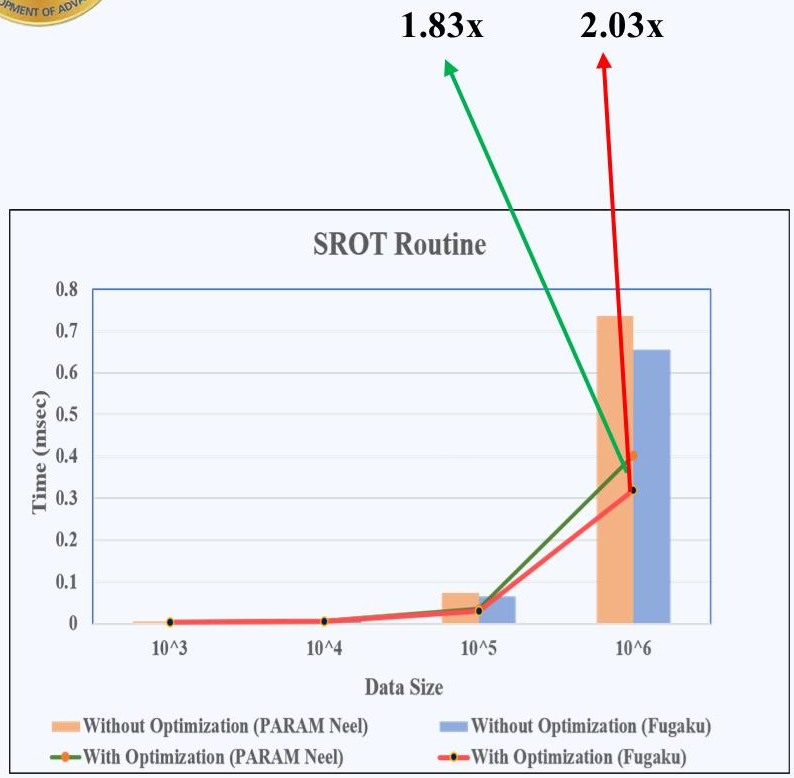


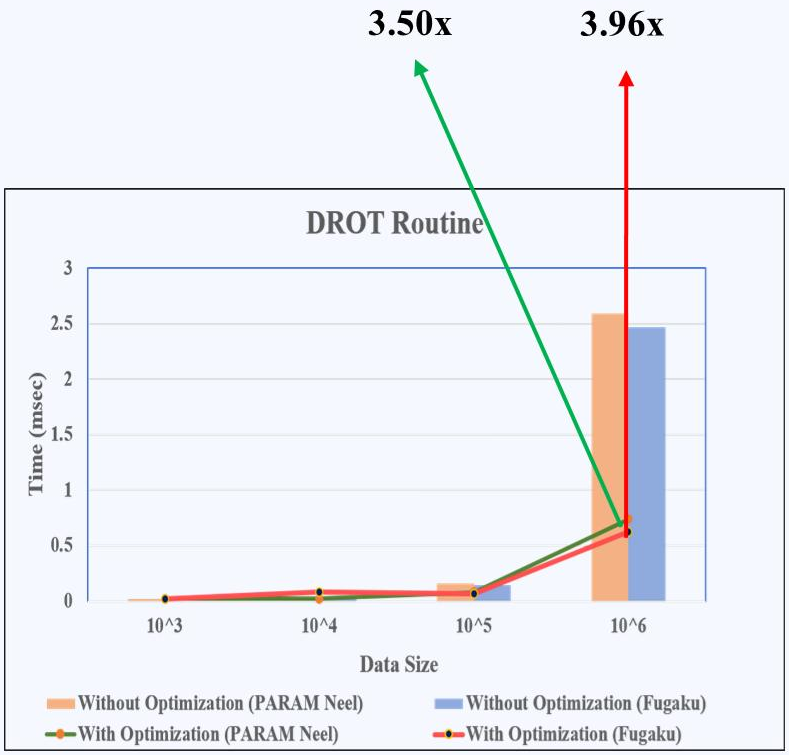
Performance Analysis Of SCAL Routine





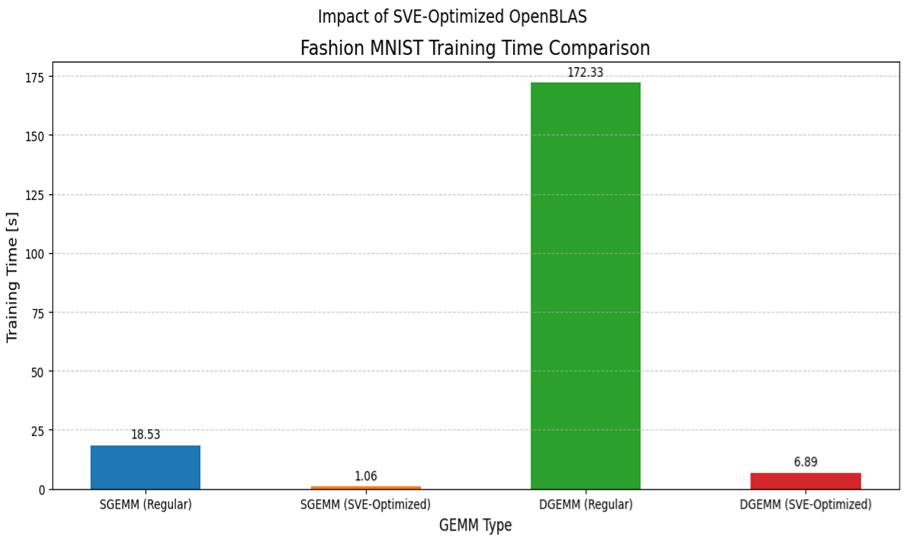
Performance Analysis Of ROT Routine





Accelerating NumPy with SVE

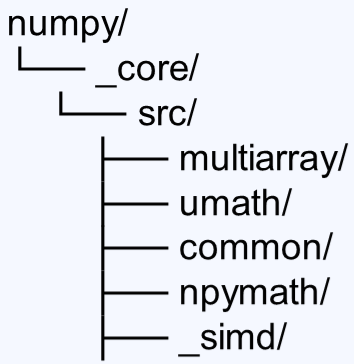
Accelerating AI Application with SVE



- We used an SVE-optimized OpenBLAS backend to accelerate Fashion MNIST model training on ARM, achieving approximately 25× speedup, showcasing the power of Scalable Vector Extensions (SVE) in AI workloads.

Accelerating AI Application with SVE

- Motivated by these results, we are extending SVE optimization into core NumPy, a heavily used foundational library in AI applications, to bring more performance gains across broader workloads.
- NumPy Structure:



NumPy SVE Integration – Workflow Overview

- SVE support in NumPy involves runtime feature detection, dispatch logic updates, and custom kernel execution.
- **Key Workflow Stages:**
  1. User Interaction / Dispatch Initiation
  2. Dispatch Selection & CPU Feature Detection
  3. Data Type & Python C API Layer
  4. SIMD Abstraction
  5. SVE Kernels & Memory Handling

The work presented here is part of the paper titled "***Optimizing NumPy with SVE Acceleration on ARM Architectures***,"which was presented at the ***International Conference on Parallel Processing*** (ICPP) 2025, held in San Diego, USA

I. User Interaction & Dispatch Initiation

- NumPy internally exposes SIMD targets through numpy.core._simd.
- Our modification adds a new '**SVE**' target to this registry.
- Enables explicit invocation of SVE-optimized functions for testing or benchmarking.

II. Dispatch Selection & CPU Detection

- **At runtime:**
  - Uses npy_cpu_init() to probe the CPU vectorization capabilities.
- **For ARM systems:**
  - It uses getauxval(AT_HWCAP) & HWCAP_SVE to check for SVE support.
  - If the SVE hardware flag is detected, NumPy sets npy__cpu_have[NPY_CPU_FEATURE_SVE] = 1;
- This activates the '**SVE**' target and ensures optimized kernels are used.
  - The dispatcher logic dynamically selects the best implementation: **SVE > NEON > Scalar baseline**
- This ensures **maximum performance portability** without recompilation.

III. Data Type & Python C API Layer

- **Data Input:** NumPy functions start with standard array inputs (e.g., np.float32).
- **Python C API Layer:**
  - Acts as the gateway from Python to NumPy's SVE-enhanced backend.
  - Parses Python arguments, validates types/shapes.
  - Converts Python-level ndarrays to C-level structures:
    - Data buffers (PyArray_DATA)
    - Dimensions (PyArray_DIMS)
    - Strides (PyArray_STRIDES)
  - Extracts raw pointers and dimensions.
  - Manages outer-level loops and delegates to vectorized inner kernels.
- **These are passed to low-level C routines and then to SVE kernels.**

IV. SIMD Abstraction (npyv_*)

- **npyv_ API:** NumPy abstracts SIMD logic through a unified npyv_* API, enabling portable vector operations across six supported SIMD architectures such as NEON, ASIMD, and SVE.
  - Translated into architecture-specific instructions via inline functions and macros.
- **SVE-Specific Vector Definitions:**
  - Located in: numpy/core/src/common/simd/sve/
  - Central header (sve.h) wraps Arm's official intrinsics from arm_sve.h.
  - Defines NumPy-specific vector types (e.g., npyv_f32 → svfloat32_t).
  - Uses conditional compilation to isolate SVE logic.
- **Mapping npyv_ API to SVE Intrinsics:**
  - Example: npyv_load_f32 → svld1_f32, npyv_mla_f32 → svmla_f32_m
  - Facilitated through preprocessor macros (e.g., NPYV_IMPL_SVE_OP) and inline wrappers.

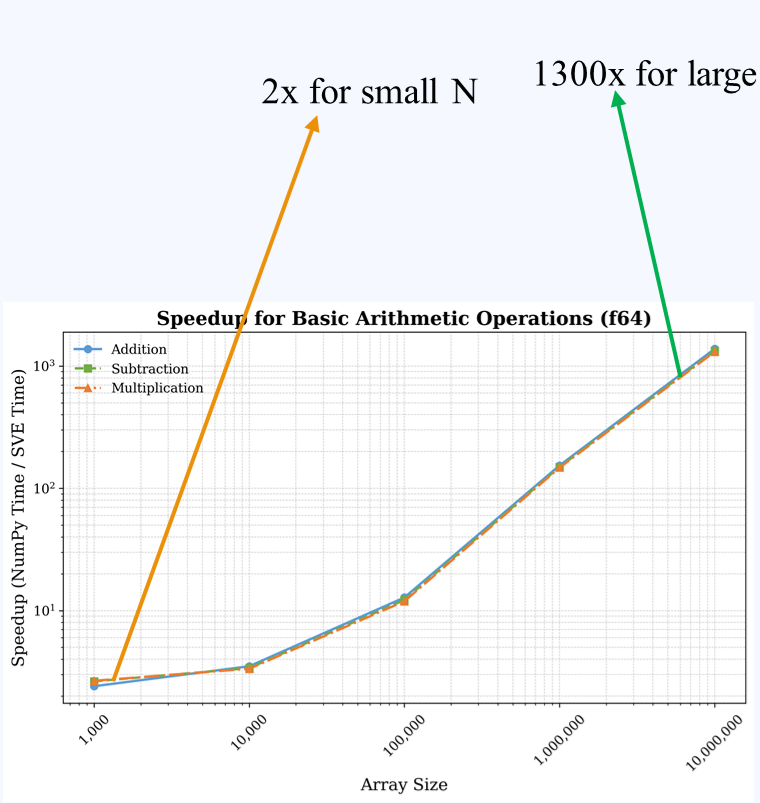
V. SVE Kernels & Memory Handling

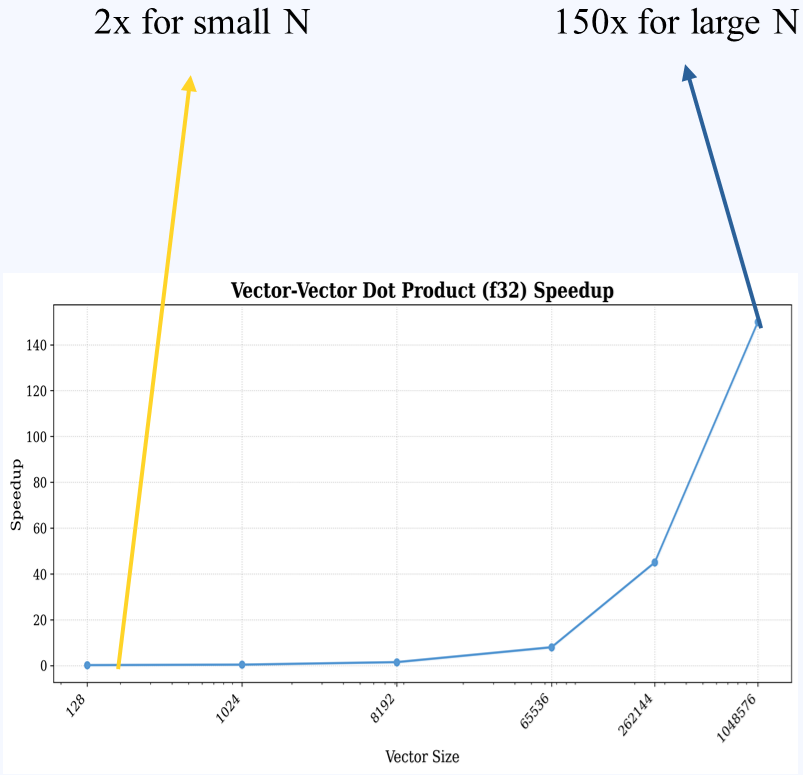
- **Hand-tuned SVE Kernels:** Numerical computations use Arm C Language Extensions (ACLE) intrinsics.
- **Vector Length Agnostic (VLA) Loops:** Core pattern uses svwhilelt (predicate masks) and svcntw() (current vector length).
- **Element-wise Ops:** Vectorized loops use svld1, svst1, and arithmetic intrinsics (e.g., svadd_f32_m) with predicate masks.
- **Accumulation:** svmla (multiply-accumulate) for computation, final reduction with svaddv.
- **Memory Handling:** SVE kernels leverage vector load/store instructions with predicate masks for efficient data handling from pre-allocated NumPy-aligned memory, ensuring optimal performance even at loop tails.

Experimental Setup – System Comparison

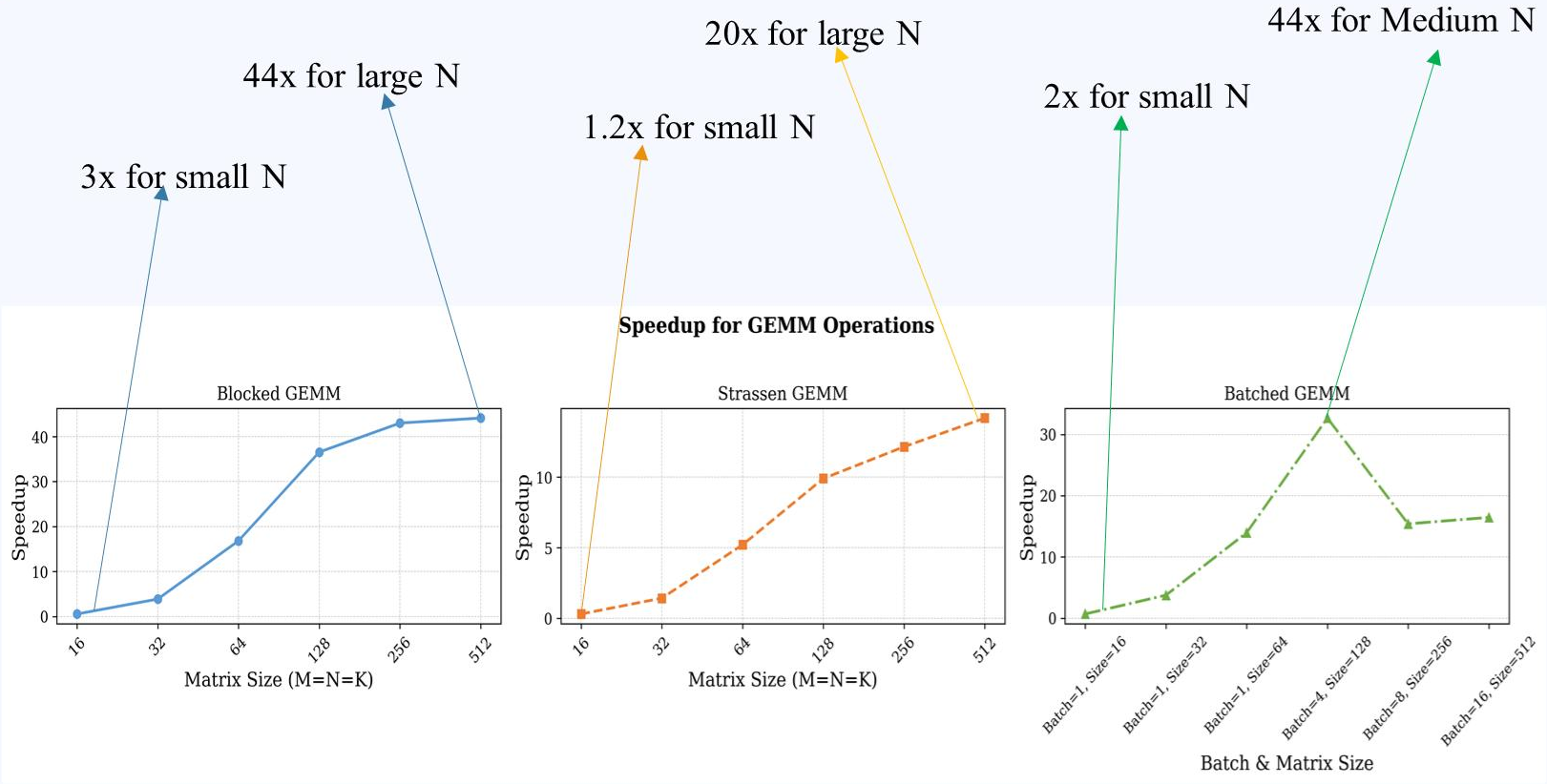
Info	PARAM Neel
Clock speed	1.8GHz
Cores	48
Vectorization	SVE 512 bits
Memory	32 GB HBM2
Compiler	GCC-12.2

Performance: Basic Arithmetic and Dot Product Speedup



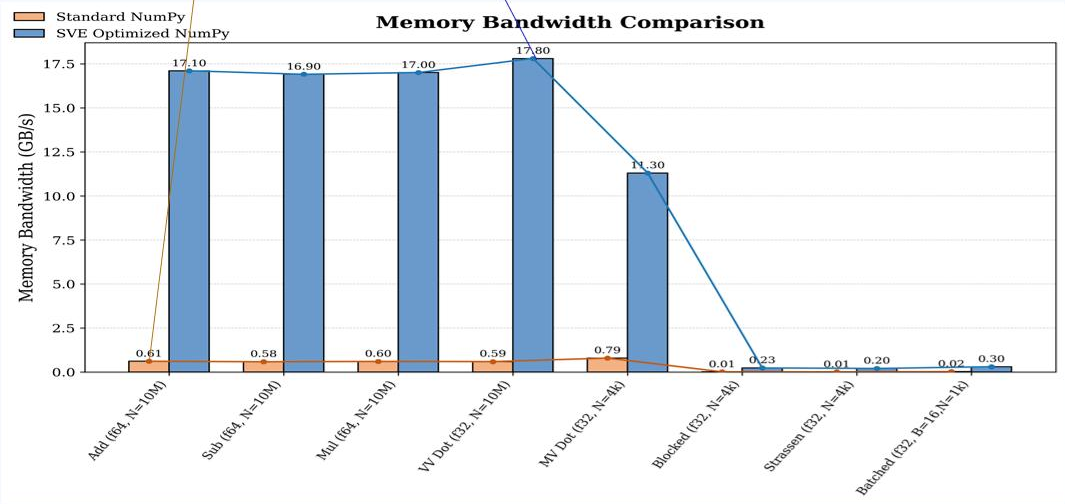


Performance: GEMM Operations Speedup



Performance: Memory Bandwidth Comparison

0.6 GB/s to 17 GB/s



Conclusion

- Results show significant performance improvements due to effective code vectorization:
  - Enhancements for Level 2 BLAS routines range from 7x to 13x.
  - Enhancements for Level 1 BLAS routines range from 1.80x to 4x.
- Performance Highlights (vs. Standard NumPy on A64FX):
  - Element-wise arithmetic: Up to 1300× speedup.
  - Vector-Vector Dot: Up to 150× speedup.
  - Matrix-Vector Dot: Up to 14× speedup.
  - Blocked/Batched GEMM: Up to 44× speedup.
  - Strassen GEMM: Up to 20× speedup.


भारत संगणन विकास केंद्र
सी डैक
CDAC
॥ज्ञानादेव तु कैवल्यम्॥
CENTRE FOR DEVELOPMENT OF ADVANCED COMPUTING

Thank You