M. H. Biniaz (mbiniaz@gwdg.de)
S. Madde (shrinath.madde@gwdg.de)

Hackathon:Arm SME boosting performance of AI and other applications on CPUs

Who am I?

(M.) Hossein Biniaz

- **AI Researcher** at **AG Computing GWDG** since 2021.
- **Research interests:**
  - Scientific algorithms
  - Distributed computing
  - Federate LLMs
  - Computational finance
- **Mission statement:**
  - Make mathematical stuff run faster on hardware.
  - Make AI available to everyone via software.

Who am I?

Shrinath Madde

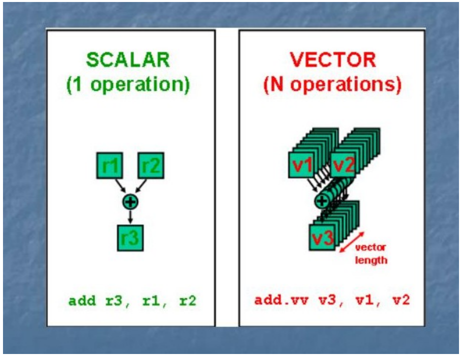
- **Master's student of Applied Data Science** at the **University of Göttingen**.
- **Key Interests:**
  - Machine Learning
  - Data Engineering
  - High-Performance Computing
- **Career Goal:**
  - To apply data science to solve real-world problems efficiently.

Table of contents

Introduction: The Need for Speed

- **What is Vector Processing?**
  - A class of parallel computing where a single instruction operates on multiple data points simultaneously (SIMD).
  - In contrast to scalar processing, where one instruction operates on one piece of data.
- **Why it's Important:**
  - **Efficiency:** Greatly accelerates tasks with repetitive operations on large datasets.
  - **Performance:** Boosts performance in multimedia, scientific computing, and machine learning.
  - **Power Savings:** Processing more data with fewer instructions can lead to lower power consumption.

Image generated with ChatGPT (AI-generated)



Scalar vs. Vector (SIMD) Processing

Scalar vs. Vector: A Tale of Two Additions

Scalar Addition (The Old Way)

```
// Goal: C[i] = A[i] + B[i];

// Must loop through each
// element one by one.

for (int i=0; i<4; i++) {
 C[i] = A[i] + B[i];
}
```

Note: This requires four separate "add" operations and loop overhead.

Vector Addition (The SIMD Way)

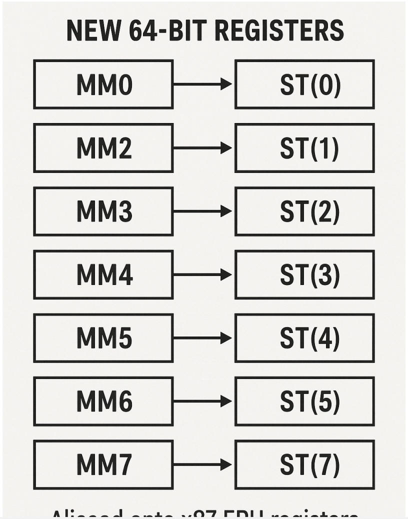
```
// Goal: Add two vectors.
// (Using Intel SSE intrinsics)

// Load 4 elements at once
__m128 vecA = _mm_load_ps(A);
__m128 vecB = _mm_load_ps(B);

// Add all 4 elements
// in one instruction
__m128 vecC = _mm_add_ps(vecA, vecB);
```

Note: This performs all four additions in a single, highly efficient instruction.

x86 SIMD: MMX (MultiMedia eXtensions)

- Introduced by **Intel in 1997** with Pentium MMX.
- **Purpose:** Accelerate multimedia and communication tasks.
- **Key Features:**
  - 8 new **64-bit registers** (MM0-MM7).
  - For example, you could pack eight 8-bit integers into one register. This meant you could, in theory, perform eight calculations at the same time.
  - Registers aliased onto x87 FPU registers.The clever, but also problematic

Image generated with ChatGPT (AI-generated)



x86 SIMD: MMX – Limitations

- **Integer-only:** Did not support floating-point SIMD operations.
- **FPU State Sharing:** Register aliasing with the x87 FPU caused significant **context switching overhead** when switching between MMX and floating-point operations.
- **Limited Vector Width:** The 64-bit vector width was quickly outgrown., which was relatively small compared to later SIMD extensions.

SIMD Feature Comparison

Feature	**SSE**	**AVX**	**AVX2**	**AVX-512**
Intro Year	~1999	~2011	~2013	~2015
Vector Width	128-bit	256-bit	256-bit	512-bit
Registers	8	16	16	32
3-operand?	× (2-op)	✓	✓	✓
Integer SIMD	Partial (SSE2+)	Float only	Full	Full
FMA (Fused Multiply-Add)	×	×	✓	✓
Masking/ Predication	×	×	×	✓ (per-lane)
Power Impact	Low	Higher	Higher	● Very High (downclock)
Use Case	Graphics, Basic Math	HPC, General Vector	HPC, ML Inference	HPC, ML, Datacenter

SIMD Features

- **3-Operand Instructions**
  - This is a huge deal for programming efficiency. Instead of an operation like `A = A + B`, which overwrites and *destroys* your original 'A' data, you can perform a **non-destructive** operation like `C = A + B`.
  - This keeps your original sources ('A' and 'B') available for other calculations, making code cleaner and more flexible.
- **Fused Multiply-Add (FMA)**
  - FMA combines a multiplication and an addition into a single, ultra-fast hardware instruction: `Result = (A * B) + C`.
  - This is faster and more precise than doing the two operations separately. It has become a cornerstone of modern **High-Performance Computing (HPC)** and **Machine Learning**.
- **Masking (Predication)**
  - Masking lets you selectively apply an operation to only the specific data elements you care about within a vector.

PowerPC: AltiVec (Velocity Engine)

- **Introduced:** 1998 by the AIM alliance (Apple, IBM, Motorola). First appeared in the **PowerPC G4**.
- **A RISC Approach:** A powerful and flexible SIMD instruction set for the PowerPC architecture.
- **Key Features:**
  - A dedicated set of 32 **128-bit vector registers**.
  - Rich instruction set supporting both integer and floating-point data.
  - A flexible **permute engine** (‘vperm‘), which was a standout feature for complex data shuffling.
- **Impact:**
  - A key selling point for Apple’s Power Mac G4/G5 systems (especially in creative apps like Photoshop).
  - Showcased the power of a well-designed SIMD architecture beyond basic multimedia.
  - Found in many game consoles (Xbox 360, PS3) and embedded systems.

ARM NEON: SIMD for the Mobile Revolution

- **Introduced:** With the ARM Cortex-A8 processor, as part of the ARMv7 architecture.
- **SIMD for Mobile:** Brought powerful SIMD capabilities to the world of mobile and embedded devices.
- **Key Features:**
  - **128-bit Registers:** A dedicated register file with 128-bit registers.
  - **Flexible Data Processing:** Supports a wide range of integer and single-precision floating-point operations.
  - **Tightly Coupled with ARM Core:** Designed for efficient and low-power media processing.
- **Impact:**
  - Crucial for the **smartphone revolution** (video, gaming, imaging).
  - Now a standard feature in modern ARM Cortex-A series processors.
  - Key technology for mobile, automotive, and server applications.

The Challenge: A World of Fixed-Width SIMD

- **The Problem of Diversity:** Powerful as they are, architectures like NEON and AVX share a common challenge.
- **Code is "Locked-in":** A developer must write code for a **fixed vector width**.
  - Your NEON code is written for 128-bit registers.
  - Your AVX2 code is for 256-bit registers.
- **The Future-Proofing Dilemma:** What happens when the next generation of hardware has a wider vector unit?
  - The software doesn't automatically benefit.
  - It forces developers to **rewrite or recompile** code to take advantage of new hardware capabilities.

The Solution: ARM SVE - Vector Length Agnostic (VLA) Programming

- ARM's brilliant solution is **Vector Length Agnostic (VLA)** programming.
- **Write Once, Scale Anywhere:** Developers write a single stream of SVE code. The code itself doesn't know or care if the hardware uses 128, 256, or even 2048-bit vectors.
- **Runtime Adaptation:** When the program runs, the hardware declares its vector length, and the SVE code **automatically scales** to use the full width available.
- This is a paradigm shift that **future-proofs software**, saves enormous development effort, and is a game-changer for HPC and diverse server environments.

The "Scalable Recipe" Analogy

Imagine writing a single recipe that just says, "for each guest, do this." At dinnertime, it

automatically works whether you have 2 or 20 guests, without you changing the recipe. **That is VLA.**

ARM SVE/SVE2: The Scalable Future

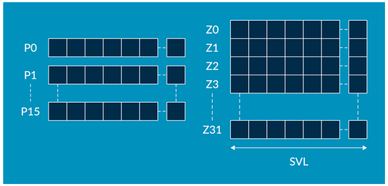
- **A New Paradigm: Vector-Length Agnostic (VLA)**
  - Introduced with ARMv8.2-A (SVE) and ARMv9 (SVE2).
  - Vector length is not fixed; can be from 128 to 2048 bits.
- **Key Architectural Features**
  - **Scalable Vector Registers (Z0-Z31):** The core data registers, whose width is defined by the hardware implementation.
  - **Scalable Predicate Registers (P0-P15):** Used for masking, enabling complex conditional execution on a per-lane basis.
  - **Gather-Load / Scatter-Store:** Instructions to efficiently handle non-contiguous data in memory, a common bottleneck.
  - **First-Fault Register (FFR):** A special predicate register that enables speculative execution of loops, improving performance.
- **Evolution from SVE to SVE2**
  - **SVE:** Primarily focused on HPC and scientific computing.
  - **SVE2:** A superset that extends SVE to accelerate a wider range of workloads, including DSP and multimedia (tasks previously handled by NEON).

SVE Feature Focus: Predicate Registers (Masking)

- **What are they?** Predicate registers (P0-P15) are a set of special registers that hold a "mask" of true/false values.
- **How it Works: A Concrete Example**
  - Assume a 128-bit SVE system. A Z register holds 16 bytes.
  - A corresponding P register is 16 bits long – one bit for each byte.
  - Let's say we have a mask in P0:

    `P0 = 1111000011110000`
  - When we run an instruction like:
    `ADD Z0.B, P0/M, Z0.B, #5`
  - The mask directly controls the operation:
    - A **1** bit enables the addition for the corresponding data element (in this case, a byte).
    - A **0** bit disables it.

Image taken from developer.arm.com



Predicate registers (P0-P15) correspond to the scalable vector registers (Z0-Z31).

- This direct bit-to-byte link is what provides incredibly **fine-grained control**.

ARM SME: Hardware Acceleration for Matrices

- **Purpose:** Builds on SVE2 to dramatically accelerate matrix operations, targeting **AI and Machine Learning** workloads.
- **Core Architectural Additions:**
  - **The ZA Register:** A massive, 2D matrix register file on the CPU. Think of it as a hardware "tile" or spreadsheet for holding matrices.
  - **Streaming SVE Mode:** A new execution mode designed specifically for matrix and streaming data operations.
- **Key Operation: Outer Product**
  - The fundamental operation is the matrix outer product.
  - Instructions like FMOPA take two vectors from the Z registers, multiply them, and accumulate the result directly into the ZA tile.
- **Impact: A Leap in Performance**
  - Allows ARM CPUs to perform matrix math at speeds previously reserved for specialized accelerators like GPUs.

BFloat16 (BF16): The Data Type Built for AI

- **The Problem with FP32:** While accurate, the standard 32-bit float is too large for AI. It consumes significant memory and bandwidth, slowing down training and inference.
- **The 16-bit Trade-Off:**
  - **FP16:** Is small, but sacrifices **range**, making it prone to errors (overflow/underflow) during AI training.
  - **BF16:** The winning solution. It also uses 16 bits but keeps the **same high range as FP32** by sacrificing some precision.

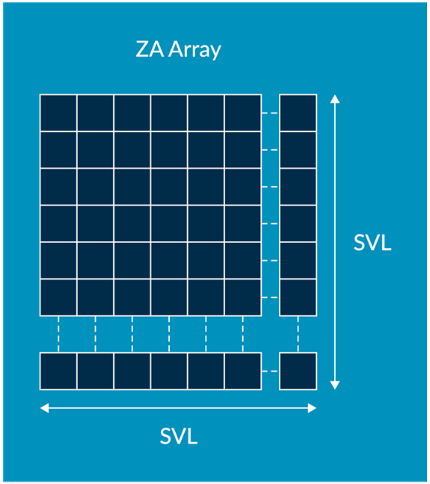
Format	Total Bits	Exponent Bits	Mantissa Bits	Range	Precision
FP32	32	8	23	High	High
FP16	16	5	10	Low	Low
BF16	**16**	**8**	**7**	**High**	**Medium-Low**

- **Conclusion:** AI models are resilient to lower precision but require a high range. BF16 provides this stability at half the memory cost of FP32. Modern ARM processors have native hardware support to accelerate it.

SME Architectural Feature: The ZA Array

- **A 2D Hardware Matrix:**
  - The heart of SME is the **ZA Array**, a physical, square-shaped register file built directly into the CPU core.
  - **Analogy:** If normal vector registers are a single row, the ZA array is a large grid—perfect for the 2D structure of a matrix.
- **Its Purpose: A High-Speed Accumulator**
  - Its main job is to hold the intermediate and final results of matrix multiplication.
  - It acts as a high-speed "scratchpad" to accumulate thousands of calculations without constantly writing to slow main memory.

Image taken from developer.arm.com



SVL

ZA Array: Scalability and Access

- **Scalable by Design**
  - Like SVE, the ZA array's size is not fixed. It is defined by the hardware's **Streaming Vector Length (SVL)**.
  - It is always a square grid that is **SVL-bytes by SVL-bytes**.
    - A 128-bit (16-byte) SVL means a 16x16 byte grid.
    - A 512-bit (64-byte) SVL means a massive 64x64 byte grid.
- **Accessed via "Tiles"**
  - A programmer works with smaller, square sections of the grid called **tiles**.
  - The hardware provides different "views" of the array, allowing it to be treated as being made up of tiles of different data formats (e.g., 8-bit integers, 32-bit floats).

SME Architectural Feature: ZA Tiles

- A **ZA Tile** is a logical, square sub-matrix within the single, physical ZA array. It is the basic unit for matrix operations.
- The number of available tiles and the size of each tile (in elements) depends on the **data type** being used.
- This allows the same hardware to be used flexibly for different precisions:

Data Type	**# of Tiles**	**Tile Names**	**Tile Size (256-bit SVL)**
8-bit (.B)	1	`ZA0.B`	32x32 elements
16-bit (.H)	2	`ZA0.H-ZA1.H`	16x16 elements
32-bit (.S)	4	`ZA0.S-ZA3.S`	8x8 elements
64-bit (.D)	8	`ZA0.D-ZA7.D`	4x4 elements
128-bit(.Q)	16	`ZA0.Q-ZA15.Q`	2x2 elements

- Programmers simply select a tile by name (e.g., `ZA1.D`), and the hardware handles the underlying size and layout.

SME: Accessing ZA Tiles

Fine-Grained Tile and Slice Operations

- **Whole-Tile Access**
  - Tiles are accessed directly for full-tile operations (e.g., initializing, loading, storing, computing).
  - Example: `STR ZA0.S, [Xn]` stores the full tile to memory.
- **Tile Slice Access**
  - Access a row or column of a tile:
    - `ZA0H.S[3]` $\rightarrow$ Row 3 (horizontal slice)
    - `ZA0V.S[5]` $\rightarrow$ Column 5 (vertical slice)
  - Enables fine-grained vector-level operations.
- **Efficient Storage**
  - Internally interleaved layout optimizes bandwidth and access patterns.

Conceptual ZA Usage

```
% Activate Streaming SVE mode
SMSTART ZA

% Load matrix tiles from memory
SMLOAD ZA0.S, [A]
SMLOAD ZA1.S, [B]

% Multiply-accumulate:ZA2 += ZA0 * ZA1
SMMLA ZA2.S, ZA0.S, ZA1.S

% Store result tile to memory
SMSTORE [C], ZA2.S

% Deactivate Streaming SVE mode
SMSTOP
```

VLA Programming: A Code Example

The Old Way: Fixed-Width (e.g., AVX2)

```
// Goal: C[i] = A[i] + B[i];
// This code is LOCKED to 256-bit vectors.

void add_arrays_avx2(float* C, float* A, float* B, long n) {
    long i = 0;
    // Process 8 floats (256 bits) at a time
    for (; i <= n - 8; i += 8) {
        __m256 vecA = _mm256_load_ps(&A[i]);
        __m256 vecB = _mm256_load_ps(&B[i]);
        __m256 vecC = _mm256_add_ps(vecA, vecB);
        _mm256_store_ps(&C[i], vecC);
    }
    // (Handle remaining elements less than 8...)
}
```

Problem: If a new CPU has 512-bit registers, this code still only processes 8 floats at a time. It doesn't get faster without a rewrite.

The New Way: VLA (e.g., ARM SVE)

```
// Goal: C[i] = A[i] + B[i];
// This code works on ANY vector length.
void add_arrays_sve(float* C, float* A, float* B, long n) {
    long i = 0;
    svbool_t pg; // The predicate (mask)
    // "While i is less than n..."
    // svwhilelt_b32 creates a mask that automatically
    // fits the hardware's vector length.
    while (pg = svwhilelt_b32(i, n), svptest_first(pg)) {
        svfloat32_t vecA = svld1_f32(pg, &A[i]);
        svfloat32_t vecB = svld1_f32(pg, &B[i]);
        svfloat32_t vecC = svadd_f32_x(pg, vecA, vecB);
        svst1_f32(pg, &C[i], vecC);
        // Increment by the number of elements
        // we just processed (the vector length).
        i += svcntw();
    }
}
```

Solution: This single piece of code will automatically use the full width (128, 256, 512-bit, etc.) of whatever SVE machine it runs on. No rewrite needed.

What is Streaming SVE (SSVE) Mode?

A Dedicated Execution Mode for SME

What is Streaming SVE Mode (SSVE)?

- A **special execution mode** introduced by SME.
- Its only job is to make the heavy math in **AI and scientific computing** run incredibly fast.
- It does this by using a small set of hyper-efficient instructions, making it much better at this one specific task than the standard, more flexible SVE2 mode.

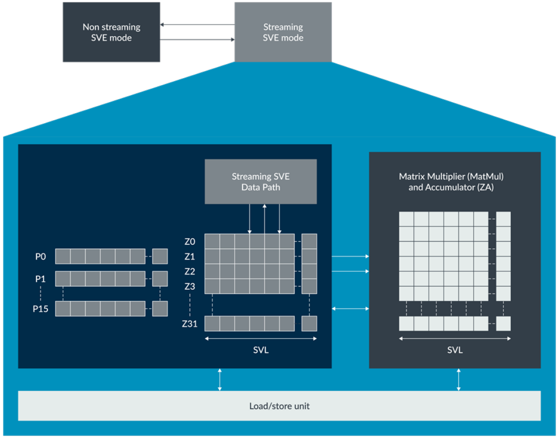


Image taken from developer.arm.com

Streaming SVE (SSVE) Features Mode Management

Key Aspects and Conceptual Workflow

Key Features of SSVE Mode:

- **Activates ZA tile registers:** SMSTART enables ZA access.
- **Distinct Streaming Vector Length (SVL):** Can differ from non-streaming VL; often hardware-optimized for size.
- **Optimized for data streaming:** Hardware handles matrix operations (e.g., SMMLA) efficiently with minimal software intervention.
- **Subset of SVE2 instructions:** Focuses on high-throughput streaming math (no complex predication, gather/scatter).

Enter Streaming SVE

```
SMSTART [ZA]
```

Conceptual SME Workflow

```
SMLOAD ZA0.S, [A] // Load A
SMLOAD ZA1.S, [B] // Load B
SMMLA ZA2.S, ZA0.S, ZA1.S // ZA2+=ZA0*ZA1
SMSTORE [C], ZA2.S // Store C
```

Exit Streaming SVE

```
SMSTOP [ZA]
```

The Math Concept: What is an Outer Product?

- An **outer product** is a fundamental operation in linear algebra where you multiply two simple lists of numbers (vectors) to create a full 2D grid (a matrix).
- **Simple Example:**
  - Take a column vector ‘[2, 3]‘ and a row vector ‘[10, 20]‘.
  - The outer product multiplies every element in the column by every element in the row.

$$\begin{bmatrix} 2 \\ 3 \end{bmatrix} \otimes \begin{bmatrix} 10 & 20 \end{bmatrix} = \begin{bmatrix} 2 \times 10 & 2 \times 20 \\ 3 \times 10 & 3 \times 20 \end{bmatrix} = \begin{bmatrix} 20 & 40 \\ 30 & 60 \end{bmatrix}$$

- This operation is a key building block for performing a full matrix-matrix multiplication.

SME: Building a Matrix Multiplication

Conceptual Example: C += A * B

- **Step 1: The First Outer Product**
  - Load the **first column** of matrix A into a Z register.
  - Load the **first row** of matrix B into another Z register.
  - Use FMOPA to compute their outer product and add it to the ZA tile.
- **Step 2: The Second Outer Product**
  - Load the **second column** of matrix A.
  - Load the **second row** of matrix B.
  - Use FMOPA again. The hardware adds this new outer product to the result already in the ZA tile.
- This "calculate and accumulate" process repeats in a loop until the final matrix C is complete in the ZA tile.

Conceptual Assembly Loop

```
// Enter Streaming Mode & clear ZA tile
SMSTART ZA
ZERO {ZA0.S}
// --- Loop (conceptual) ---
for k in 0..N:
   // Load column k from A into Z0
   LD1W {Z0.S}, pg, [ptr_A_col_k]
   // Load row k from B into Z1
   LD1W {Z1.S}, pg, [ptr_B_row_k]
   // Outer product and accumulate:
   // ZA0 += Z0 * Z1_transposed
   FMOPA ZA0.S, P0/M, P1/M, Z0.S, Z1.S
// --- Finalization ---
// Store result from ZA0.S to C
ST1W {ZA0.S}, pg, [ptr_C]
SMSTOP
```

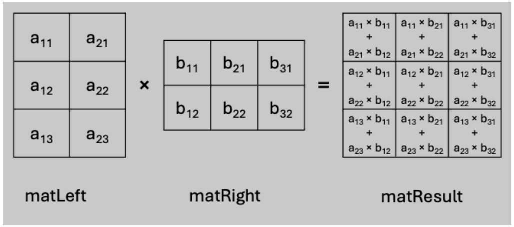
Conclusion: Key Takeaways

- **A Journey of Escalation:** We've traced the evolution of vector processing from simple, fixed-width integer units like **MMX** to incredibly complex, **scalable matrix engines** like ARM's SME.
- **The Paradigm Shift to VLA:** The most significant change has been the move from fixed-width designs to ARM's **Vector Length Agnostic (VLA)** architecture. This future-proofs software and marks a new era in processor design.
- **AI as the Primary Driver:** The recent, rapid innovation in this space—especially with **SME**, dedicated matrix hardware, and new data formats like **BFloat16**—is overwhelmingly driven by the massive computational demands of AI and Machine Learning.
- **The Future is Co-Design:** The path forward is no longer just about wider vectors. It's about the deep, intentional co-design of hardware and software to solve specific, complex problems with maximum efficiency.

Vanilla matrix multiplication

What is a vanilla matmul?

- The vanilla matrix multiplication operation takes two input matrices, A [Ar rows x Ac columns] and B [Br rows x Bc columns], to produce an output matrix C [Cr rows x Cc columns].
- The operation consists of iterating on each row of A and each column of B, multiplying each element of the A row with its corresponding element in the B column then summing all these products


a_{11} a_{21} a_{12} a_{22} a_{13} a_{23}
×
b_{11} b_{21} b_{31} b_{12} b_{22} b_{32}
=
$a_{11} \times b_{11} + a_{21} \times b_{12}$ | $a_{11} \times b_{21} + a_{21} \times b_{22}$ | $a_{11} \times b_{31} + a_{21} \times b_{32}$
$a_{12} \times b_{11} + a_{22} \times b_{12}$ | $a_{12} \times b_{21} + a_{22} \times b_{22}$ | $a_{12} \times b_{31} + a_{22} \times b_{32}$
$a_{13} \times b_{11} + a_{23} \times b_{12}$ | $a_{13} \times b_{21} + a_{23} \times b_{22}$ | $a_{13} \times b_{31} + a_{23} \times b_{32}$
matLeft
matRight
matResult

https://learn.arm.com/learning-paths/cross-platform/multiplying-matrices-with-sme2/3-vanilla-matmul/

Vanilla matrix multiplication

- **What is a vanilla matmul?**
  - The vanilla matrix multiplication operation takes two input matrices, A [Ar rows x Ac columns] and B [Br rows x Bc columns], to produce an output matrix C [Cr rows x Cc columns].
  - The operation consists of iterating on each row of A and each column of B, multiplying each element of the A row with its corresponding element in the B column then summing all these products.

vanilla matmul

```
void matmul(uint64_tM, uint64_tK, uint64_tN, constfloat ∗
        restrictmatLeft, constfloat ∗
        restrictmatRight, float ∗ restrictmatResult){
for (uint64_tm = 0; m < M; m + +){
for (uint64_tn = 0; n < N; n + +){
float acc = 0.0;
for (uint64_tk = 0; k < K; k + +)
acc += matLeft[m ∗ K + k] ∗ matRight[k ∗ N + n];
matResult[m ∗ N + n] = acc; } } }
```

macc to load ratio

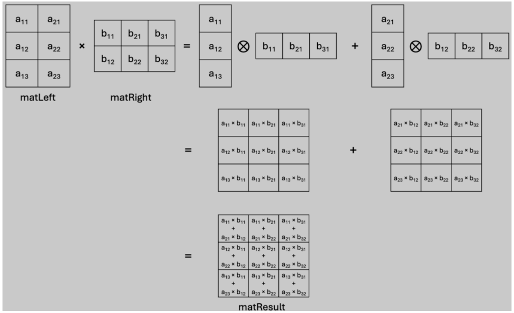
- **What is a macc:load ratio**
  - translates to 1 multiply-accumulate, which is also known as macc, for two loads (matLeft[m * K + k] and matRight[k *N + n]).
  - It therefore has a 1:2 macc to load ratio.
  - From a memory system perspective, this is not effective, especially since this computation is done within a triple-nested loop, repeatedly loading data from memory.
  - To exacerbate matters, large matrices might not fit in cache. In order to improve the matrix multiplication efficiency, the goal is to increase the macc to load ratio, which means to increase the number of multiply-accumulate operations per load.

core of standard matmul

```
acc += matLeft[m * K + k] * matRight[k * N + n];
```

Outer product matrix multiplication

- **What is a outer prod matmul?**
  - matLeft (3 rows, 2 columns) by matRight (2 rows, 3 columns), decomposed as the **sum of the outer products**:
- **About transposition**
  - Matrices are laid out in row-major order in memory: Loading row-data from memory is efficient (contiguous data)
  - Caches are loaded row by row, data prefetching is simple - just load the data from current address + sizeof(data).
  - not the case for loading column-data from memory system.
  - gather load of sve2 doesnt work with sve-streaming (=sme)

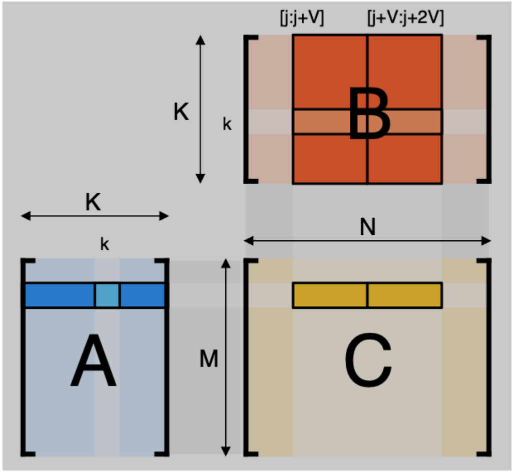

matLeft
matRight
matResult
a_{11} a_{21} a_{12} a_{22} a_{13} a_{23}
b_{11} b_{21} b_{31} b_{12} b_{22} b_{32}
$a_{11} \times b_{11}$ + $a_{21} \times b_{12}$

https://learn.arm.com/learning-paths/cross-platform/multiplying-matrices-with-sme2/3-vanilla-matmul/

Outer product matrix multiplication

The case with SVE

- Each element of the left matrix ($A_{i,j}$ must be in every lane of the register and multiplied by the row starting at column j for the SVL elements
- predicates must be present to control the loop out of bounds
- Acc(umulator) must be emptied at the right time (that means after writing the result back to the shaded yellow part of the summation matrix



Arm Scalable Vector Extension and application to Machine Learning (Dan Andrei Iliescu, Francesco Petrogalli)

Outer product matrix multiplication

The case with SVE

- Each element of the left matrix ($A_{i,j}$ must be in every lane of the register and multiplied by the row starting at column *j* for the *SVL* elements
- predicates must be present to control the loop out of bounds
- Acc(umulator) must be emptied at the right time (that means after writing the result back to the shaded yellow part of the summation matrix
- c.f. *Arm Scalable Vector Extension and application to Machine Learning (Dan Andrei Iliescu, Francesco Petrogalli)*

svl intrinsics for matmul with VLA vectorization

```
void hgemm(float *C, float const *A, float const
    *B, ...)  {
for (int i = 0; i < M; ++i)
for (int j = 0; j < N; j += svcnth()) {
svfloat16_t Acc = svdup_f16(0);
const svbool_t pred_j = svwhilelt_b16(j, N);
for (unsigned long k = 0; k < K; ++k) {
const svfloat16_t A_i_k = svdup_f16(A[i * K + k]);
const svfloat16_t B_k_j = svld1(pred_j, B[k * N +
    j]);
Acc = svmla_x(pred_j, Acc, A_i_k, B_k_j); }
svst1(pred_j, C[i * N + j], Acc);
```

Resources

Nice / must-read resources:

- Arm Scalable Vector Extension and application to Machine Learning (Dan Andrei Iliescu, Francesco Petrogalli) https://developer.arm.com/-/media/Arm
- https://learn.arm.com/learning-paths/cross-platform/multiplying-matrices-with-sme2/4-outer-product/

Regarding predicates and ZA tiles (with clean link you must sign up for Arm so just google it):

- Part 1: Arm Scalable Matrix Extension (SME) Introduction https://www.google.com/url?sa=tsource=webrct=jopi=89978449url=https://community.a community-blogs/b/architectures-and-processors-blog/posts/arm-scalable-matrix-extension-introduction-p2ved=2ahUKEwio24fWsv6NAxXUSvEDHa5ABKgQFnoECB0QAQusg=AOvVaw21wPUUk9LJ ciBTR2Xgz
- Part 2: Arm Scalable Matrix Extension (SME) Instructions
- Repository: https://github.com/mhbiniaz/llama2.c-arm-sve-sme/tree/master

Q&A

Questions?