Leveraging Arm's Scalable Matrix Extension to Accelerate Matrix Multiplication Kernels

Finn Wilkinson, Jack Jones, Rahat Muneeb, Simon McIntosh-Smith

Introduction & Background

With the ever-growing interest in AI, Machine Learning, and Deep Learning, new acceleration techniques are being devised to leverage performance. One such technique is the use of matrix engines within CPUs to try and bridge the gap between CPU and GPU performance. Whilst GPUs typically dominate these kinds of workloads due to their inherent SIMD nature, they can come at a cost, namely power and data-offload overheads. As such, having matrix engines close to, or inside of, the CPU itself can provide additional performance whilst reducing these overheads. Currently, some recent CPU offerings from Intel, Apple, and IBM all have their own versions of a matrix engine, each implemented slightly differently but achieving the same goal of improved matrix multiplication performance. Although no hardware is currently available, Arm have also specified their own CPU matrix ISA extension, called the Scalable Matrix Extension (SME). Building from their Scalable Vector Extension (SVE), SME introduces new outer-product instructions and a 2-D matrix register to accelerate level 3 BLAS operations. A more recent version of the extension, SME2, adds support for inner-product and multi-vector instructions to support the acceleration of level 2 BLAS operations.

Due to the lack of available hardware, we utilise The Simulation Engine (SimEng) from the University of Bristol's High Performance Computing Group, along with the Structural Simulation Toolkit (SST) from Sandia National Laboratories [3] to simulate a hypothetical core design with an integrated SME matrix engine. This enables us to build on previous work from Wilkinson et al[4], which compared the performance of SVE and SME SGEMM. By widening the scope to SGEMM and DGEMM, we are able to more comprehensively evaluate the advantages that SME has compared to like-for-like NEON (a 128-bit SIMD extension) and SVE implementations, and how this may translate to performance gains in real workloads.

Simulation Configuration & Validation

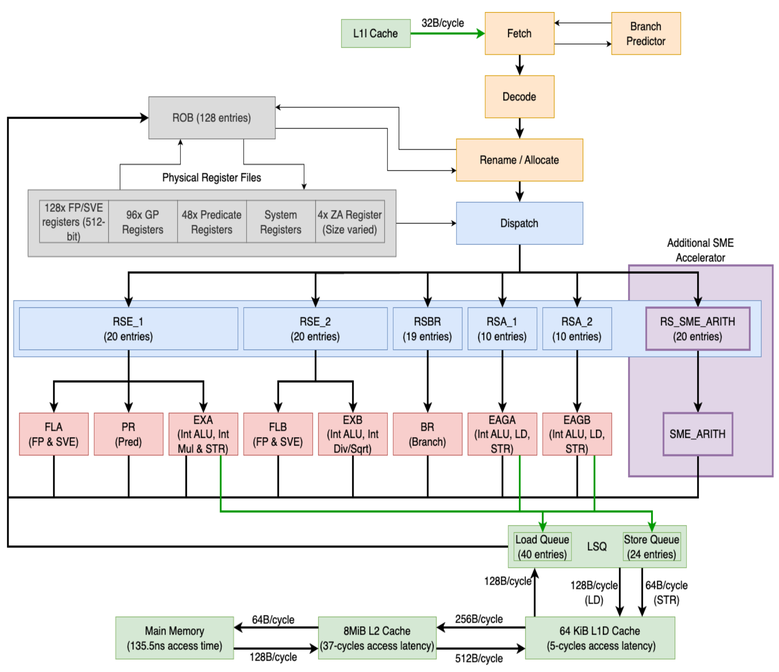

Figure 1: A64FX Core Model with added SME Accelerator

For this study, we use SimEng v0.9.4, providing us with two important new additions over previous versions: SME instruction support and SST integration. Whilst SimEng provides a framework to simulate cycle-accurate core models, SST provides a comprehensive way to simulate a complex memory hierarchy. For our experiments, we incorporated SimEng into SST as a distinct SST-Element called *sstsimeng*. This integration allows SimEng to drive the simulation of a core model, whilst the memory hierarchy simulation is handled by the *memHierarchy* module from SST-Elements.

As the basis of our hypothetical core with SME support, we chose Fujitsu's A64FX due to its implementation of SVE-512, extensive documentation (enabling a more accurate configuration), and our on-going work with RIKEN towards FugakuNext. Figure 1 depicts the A64FX core with the additional SME matrix engine; configured as a single non-blocking execution unit, additional to the existing pipeline, and its own reservation station. Given that SME loads and stores are similar to their SVE counterparts, they share the existing *EAGA* and *EAGB* units in the A64FX pipeline. All SME instruction latencies have also been configured to match their SVE counterparts.

Given our target workloads are built for a bare-metal target, we are not able to run these on in-production hardware. Hence, we instead use similar kernels in ArmPL's implementations of SGEMM and DGEMM to validate our simulation accuracy. Seen in Table 1, our accuracy is generally good, with cycle counts mostly within 15% of hardware. However, to achieve this some minor alterations to the memory configuration were required, namely reducing L2 access latency and increasing L1-L2 cache bandwidths. The former has been configured to match the L2 access latency of RIKEN's PostK Gem5 model[2], and the latter to match the A64FX's Core Memory Group total bandwidth limits[1]. These changes were made in order to improve the simulation accuracy for this study, but future releases of SimEng will look to improve the accuracy to hardware as a whole.

GEMM M=N=K	Iterations	SimEng Cycles	A64FX Cycles	% Difference
SGEMM 64	100	1,992,536	1,854,970	+7.2%
SGEMM 256	100	99,729,016	86,355,036	+14.4%
SGEMM 1024	10	611,936,990	541,571,286	+12.2%
DGEMM 64	100	3,514,885	3,714,395	-5.5%
DGEMM 256	100	199,697,264	162,705,575	+20.4%
DGEMM 1024	10	1,292,974,554	1,209,144,243	+6.7%

Table 1: Cycle comparisons of SimEng to Hardware for n iterations of the given ArmPL GEMM computation

Target Workloads

To compare SME against Arm's NEON and SVE instruction sets, we look to a workload that can be computed by both vector and matrix instruction sets. Given we place this study in the domain of high-performance computing, we assume that the use of the SVE and NEON instruction sets is fine-tuned to extract the best performance from the underlying hardware. To match these assumptions, we ensure the chosen workload is moderately optimised. The workloads under simulation are GEMM operations as defined in the level 3 BLAS routines, namely,

$$C := \alpha * A * B + \beta * C$$

with A, B, and C being matrices, and α and β being scalars kept as 1 and 0 respectively for simplicity. GEMM kernels suit our aforementioned workload characteristics well, offering a well-defined problem whose computations can be carried out efficiently with both vector and matrix instructions. The workloads used in this study were provided by Arm Ltd. and contain optimised NEON, SVE and SME implementations of SGEMM and DGEMM kernels. Each kernel computes the sum of outer products

$$C = AB = \sum_{i=0}^{K} \mathbf{a}_i \mathbf{b}_i{}^T$$

where $\mathbf{a}_i$ is column i of A, and $\mathbf{b}_i$ is row i of B. Matrix A is assumed to be pre-transposed for simplicity. Within the workload, the main GEMM computation is looped over 100 or 1,000 times, depending on problem size, to ensure caches have ample time to warm up and to amortise the overhead of any non-kernel code.

Matrix dimensions of 32, 64, and 128 (with $M = K = N$) were chosen for this study to allow for the identification of any notable performance impacts varying problem sizes may incur. We also target both SGEMM and DGEMM variants of the GEMM computation to broaden the scope of instructions in use, and to help identify any performance characteristics the data type may invoke.

SVE vs. SME: Vector Length Comparison

By comparing the cycle counts of SVE to SME, we are able to assess how SME's performance advantage over SVE grows as the vector length widens. For all results below, the SVE Vector Length (VL) matched SME's Streaming Vector Length (SVL).

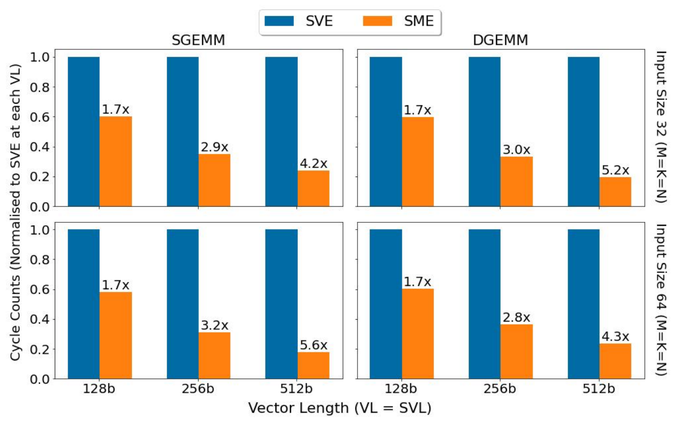

Figure 2: Performance comparison of SME vs SVE based on normalised cycles counts for smaller input sizes, lower is better

From Figure 2 and Figure 3 we can see that SME always provides a cycle advantage. With SME's compute throughput per outer-product instruction calculated as $\frac{SVL}{32}$ vectors for SGEMM and $\frac{SVL}{64}$ vectors for DGEMM, doubling the SVL will also double any theoretical peak compute SME has over SVE. However, we observe a diminishing trend in the rate of speed-up SME has over SVE as the VL increases. The larger problem size of 128 showcases an improvement in the continual performance gains over SVE up until a VL of 1024-bits.

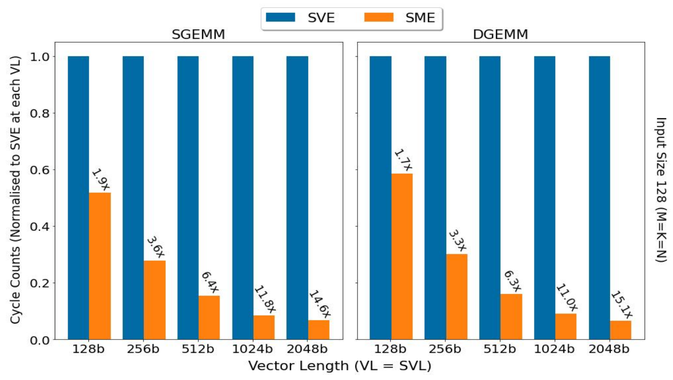

Figure 3: Performance comparison of SME vs SVE based on normalised cycles counts for the largest input size, lower is better

Within the simulated workloads, utilised SME instructions require SVE instructions to handle memory accesses. As the VL increases, the SME instructions carry out four times the amount of work whereas the SVE memory operations are only capable of a two times increase. In this scenario, SME's reliance on SVE limits its speed-up factor to be sub-peak. This limitation worsens with the increase in VL as the ratio of SVE to SME instructions increases. In Figure 3, we find the plateau in performance gains over SVE to be more aggressive at higher VLs. We attribute this to the comparably significant use of the L2 cache, coupled with a sub-optimal prefetcher; meaning data required to feed SME's compute instructions takes longer to arrive at the core and subsequently impacts their throughput.

NEON vs. SVE vs. SME

Here, we compare SME-128 to SVE-128 and NEON. In A64FX, with two FPUs, SVE and NEON can produce two result vectors per cycle. With one SME-128 unit, we can produce the equivalent of 4 vectors per cycle for SGEMM or 2 for DGEMM. Hence, we expect SME-128 to have a 2x advantage for SGEMM, and be approximately equivalent for DGEMM. However, from Figure 4 this clearly is not the case. For input sizes 32 and 64, with both SGEMM and DGEMM, our A64FX SME-128 model consistently boasts more than a 2x speed-up over NEON but never achieves a 2x speed-up over SVE-128. SME's performance then begins to drop as the problem size grows beyond L1D cache due to a sub-optimal prefetcher used in our simulation.

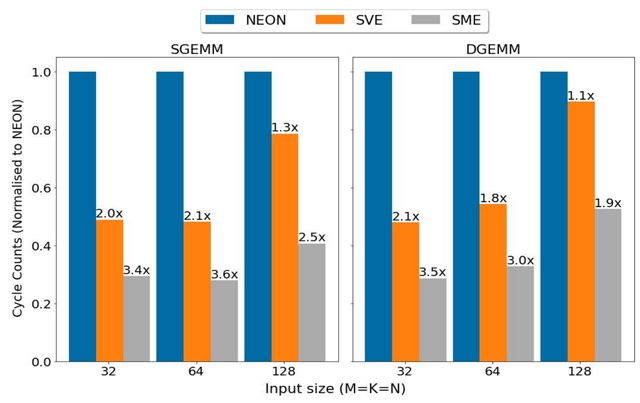

Figure 4: Performance comparison of NEON vs SVE vs SME based on normalised cycles counts, lower is better

Analysing A64FX's capacity to process the indexed FMLAs used in the NEON and SVE kernels shows NEON can achieve 0.5 IPC and SVE 1 IPC. Therefore, SME-128 should have a peak SGEMM speed-up of 8x over NEON and 4x over SVE. Hence, SME-128 is achieving roughly 50% of peak performance in our hypothetical model. This delta is due to A64FX's long vector compute and load latencies of 9 and 11 cycles respectively; leading to large pipeline bubbles as SME's outer-product instructions hold onto the limited physical registers available. For DGEMM SME-128, we expect to achieve half of SGEMM performance, but, with there being double the number of architectural ZA sub-tiles available in double-precision, we get fewer bubbles and hence better ILP; amortising the compute difference and achieving over 80% of peak performance in the 32 and 64 input sizes.

Future Work

Given the limitations A64FX's long vector pipelines imposed on the hypothetical SME implementation, conducting a further study evaluating what out-of-order resources would need to be scaled-up to achieve adequate performance, and comparing this to an SME implementation in a less specialised core such as Graviton3, would allow us to infer the micro-architectural features required to support an effective SME implementation. We also aim to evaluate SME2, which provides means to accelerate level 2 BLAS operations such as GEMV, and some of the other matrix engines on the market from vendors such as Intel, Apple, and IBM.

Acknowledgements

This work has been funded by the UKRI ASiMoV project (EP/S005072/1), and supported by Arm Ltd. via an EPSRC iCASE.

References

[1] Fujitsu Limited.
A64fx microarchitecture manual: 9. cache architecture.
pages 63-66, 30th Nov. 2022.

[2] RIKEN RCCS.
O3_postk.py.
https://github.com/RIKEN-RCCS/riken_simulator/blob\/riken/configs/common/cores/arm/O3_PostK.py.
Accessed 24th July 2023.

[3] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob.
The structural simulation toolkit.
SIGMETRICS Perform. Eval. Rev., 38(4):37–42, March 2011.

[4] Finn Wilkinson and Simon McIntosh-Smith.
An initial evaluation of arm's scalable matrix extension.
In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 135–140, 2022.