Efficient 8-bit matrix multiplication on Computational SRAM architecture

Kévin Mambu, Henri-Pierre Charles, Maha Kooli

Efficient 8-bit matrix multiplication on Computational SRAM architecture

Kévin Mambu, Henri-Pierre Charles, Maha Kooli
Université Grenoble Alpes, CEA LIST, F-38000 Grenoble, France
surname.name@cea.fr

***Abstract*—This paper proposes a micro-kernel to efficiently compute 4x4 8-bit matrix multiplication on In-Memory Computing (IMC) Architectures with 128-bit word-lines. The proposed implementation requires simple instructions with vector-data computation and could be used as a basic block to implement General Matrix Multiplication (GEMM) on 128-bit word-lines IMC architectures, using 4x4 matrix partitioning. This micro-kernel would be beneficial to domains such as image processing and computer graphics.**

***Index Terms*—in-memory computing, sram, matrix-multiplication, simd, micro-kernel**

I. Context

A. Computational SRAM architecture

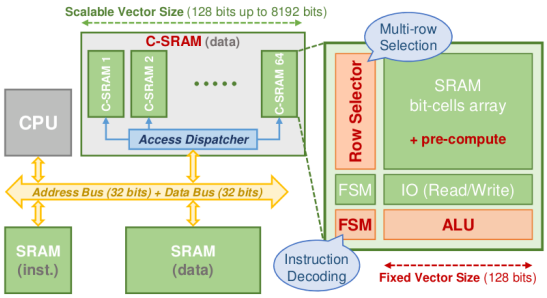


Fig. 1: Computational SRAM architecture

The Computational SRAM (C-SRAM) architecture is an SRAM-based IMC architecture allowing to perform computation directly inside the memory array. Logical and arithmetical operations are computed by an ALU slice in periphery of the bit-cell array, which allows for low-cost IMC integration in existing SRAM technologies [1] [2].

By limiting data transfers over the bus between the host CPU and the memory, this architecture offers time and energy reduction [3]. Furthermore, the C-SRAM architecture can be re-configured to perform large-vector data computation either on single SRAM tile or across multiple tiles, as seen on Fig. 1.

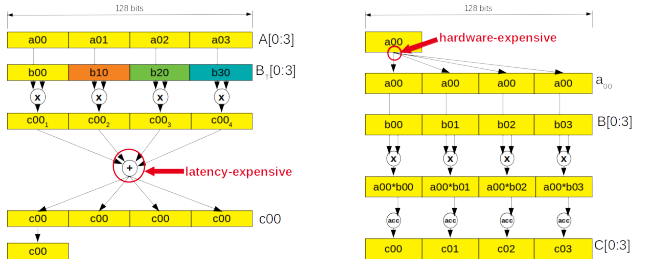
For computation to be correctly performed, the data need to be physically aligned on the same columns, inducing challenges of data placement.

This algorithm stands in the context of a C-SRAM architecture configured to 128-bit vectors.

B. Vectorization schemes for matrix multiplication

Existing works present two main methods to vectorize matrix multiplication: the first one, on Fig. 2a is to vectorize the computation of the dot-product for each element of the output matrix, and then perform an horizontal reduction by adding every element of the vector together. It requires a transposition on one of the inputs and both inputs are visited row-major [4].

The second method requires to duplicate data across one of the input vectors to compute multiple elements at once, as seen on Fig. 2b. This method has the advantage of not requiring any matrix transformation prior to computation [5].



(a) Per-row vectorization (b) Per-columns vectorization

Fig. 2: Existing vectorization schemes of matrix multiplication

Both these algorithms are efficient on large instances, when each matrix are large enough to profit from data-level parallelism. However, when using 128-bit vectors, the design of C-SRAM requires an alignment of every matrix row on the same columns, as seen on Fig. 3.

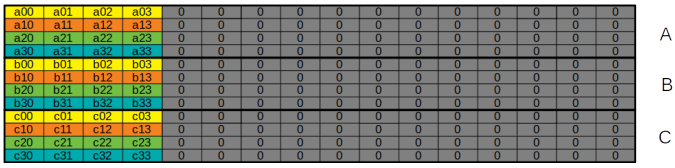


Fig. 3: Data placement required on 128-bits vectors C-SRAM.

II. Proposed implementation: Jag-and-Rotate

A. Algorithm

Our proposed implementation is an iterative process that does not require each row of the matrix to be horizontally

aligned. As such, each matrix can be entirely stored in a single vector without impeding computation.

First we transform one of the input matrix A into $A_J(0)$ such as :

$$\begin{aligned} &\text{let } 0 \leq x \leq N \text{ and } 0 \leq y \leq N : \\ &A_J(0)[x][y] = A[(x+y)\%N][x] \end{aligned} \tag{1}$$

The iterative process can be seen on Fig. 4. On each iteration $i > 0$, we rotate the rows of $A_J(i-1)$ by 1 byte to compute a partial result of C, called $C'_J(i)$, and accumulate it in a vector Acc. At each iteration, $C'_J(i)$ is re-aligned with Acc with a vector-wise rotation before being accumulated. The command for this rotation is the width of a row (`ROW_SIZE`) times i. .

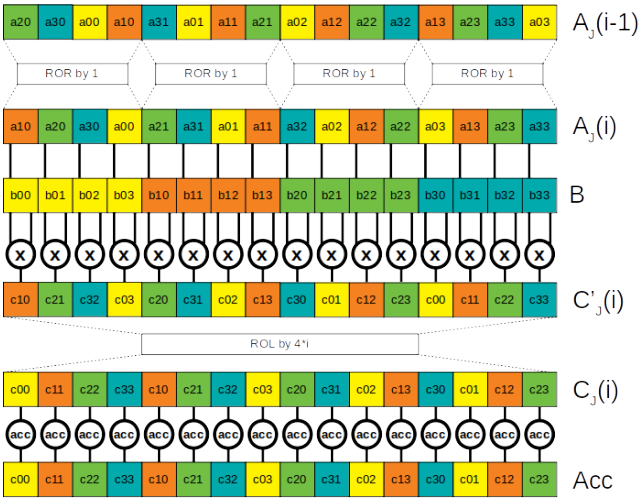


Fig. 4: Our proposed vectorization scheme

The complexity of this iterative process equals the size of the row in number of elements (here 4 elements, so 4 steps). At he last iteration the matrix C can be obtained from Acc via the following expression :

$$\begin{aligned} &\text{let } 0 \leq x \leq N \text{ and } 0 \leq y \leq N : \\ &C[x][y] = C_J[x][(x+y)\%N] \end{aligned} \tag{2}$$

B. Implementation through the features of C-SRAM

Both input (Eq. 1) and output (Eq. 2) transformations do not require any data duplication. These transformations, the 32-bit rotations on $A_J(i)$ and the vector-wise rotation on on $C'_J(i)$ can all be done using a Block Shuffle unit at byte granularity [6].

To support this vectorization scheme on larger instances the C-SRAM should possess vectors large enough to contain the matrix and 2) appropriate rotation units. Respectively, variants of this algorithm run on the 128-bit word-lines C-SRAM scale up poorly to larger matrix sizes due to the cost of inter-vector permutation, contrary to the per-row and per-columns vectorization schemes, previously presented.

C. Algorithmic evaluation

Both the per-row and per-columns vectorization schemes require row of matrix to be aligned, as illustrated in Fig. 3. In these conditions, this constraint limits the effective throughput, in terms of computations per instruction, to 4, while the theoretically maximum throughput achievable by the 128-bit word-line C-SRAM is 16 computations per instruction. One one hand, As seen on Fig. 2, the per-row scheme needs 3 instructions to compute an element, added the input transposition this makes 49 cycles. On the other hand, the per-columns scheme needs 3 times 4 instructions to compute a row, which amounts in total to 48 cycles. Our implementation, however, partially compute the entirety of the matrix at a rate of 3 instructions per iteration and delivers the result after 4 iterations. Added the two input and output transformations, our vectorization scheme multiplies 4x4 matrices in 14 cycles, while achieving the maximum throughput available on C-SRAM.

III. Perspectives

4x4 matrices in themselves are often used in image processing, to apply convolution kernels and in computer graphics as transformation matrices. Moreover, this implementation can be used as a basic block to perform 8-bit General Matrix Multiplication (`gemm`), using matrix partitioning on IMC architectures with 128-bit word-lines. Scientific libraries, generally, rest on highly-optimized routines (such as the Basic Linear Algebra Subroutines (BLAS) libraries [7]) efficient programming models to interface the applications to the hardware. Studying IMC-specific optimized routines can be a first step in designing a high-performance software stack.

IV. Conclusion

This paper proposes an efficient micro-kernel for 4x4 8-bit matrix multiplication that can be used as a basic block to perform 8-bit General Matrix Multiplication, using matrix partitioning. This micro-kernel is the beginning of a reflexion on IMC-specific micro-kernels which could be implemented in domain-specific libraries [4].

References

[1] R. Gauchi, M. Kooli, P. Vivet, J.-P. Noel, E. Beigné, S. Mitra, and H.-P. Charles, "Memory Sizing of a Scalable SRAM In-Memory Computing Tile Based Architecture," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct. 2019, pp. 166–171, iSSN: 2324-8432.

[2] M. Kooli, H.-P. Charles, C. Touzet, B. Giraud, and J.-P. Noël, "Software platform dedicated for in-memory computing circuit evaluation," in *Proceedings of the 28th International Symposium on Rapid System Prototyping Shortening the Path from Specification to Prototype - RSP '17*. Seoul, South Korea: ACM Press, Oct. 2017, pp. 43–49. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3130265.3130322

[3] R. Hartenstein and T. Kaiserslautern, "The von Neumann Syndrome," p. 7, 2007.

[4] U. D. R. Hat, "What Every Programmer Should Know About Memory," p. 114, Nov. 2007.

[5] "Coding for NEON - Part 3 matrix multiplication." [Online]. Available: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/coding-for-neon—part-3-matrix-multiplication

[6] V. Egloff, H.-P. Charles, M. Kooli, B. Giraud, J.-P. Noel, and J.-M. Portal, "Mélangeur pour la multiplication de matrice sur architecture de type calcul en mémoire," Bayonne, France, Jun. 2019.

[7] "BLAS (Basic Linear Algebra Subprograms)." [Online]. Available: http://www.netlib.org/blas/