Accelerator for Sparse Machine Learning

L. Yavits and R. Ginosar

Abstract— Sparse matrix by vector multiplication (SpMV) plays a pivotal role in machine learning and data mining. We propose and investigate an SpMV accelerator, specifically designed to accelerate the sparse matrix by sparse vector multiplication (SpMSpV), and to be integrated in a CPU core. We show that our accelerator outperforms a similar solution by 70x while achieving 8x higher power efficiency, which yields an estimated 29x energy reduction for SpMSpV based applications.

Index Terms—Sparse matrix multiplication, sparse matrix by sparse vector multiplication, SpMV, accelerator.

1 Introduction

Sparse matrix by vector multiplication (SpMV) is a frequent bottleneck in machine learning and data mining workloads. The efficient implementation of SpMV becomes even more critical when applied to big data problems.

Most research on SpMV acceleration addresses Sparse Matrix by Dense Vector multiplication (SpMDV) [7]. This paper focuses on Sparse Matrix by Sparse Vector multiplication (SpMSpV), an essential kernel in machine learning algorithms such as sparse principal component analysis (PCA), or kernelized SVM classification and regression. Sparse PCA computes a co-variance matrix from a sparse dataset. It involves multiplication of one feature sparse vector by all other feature vectors in the matrix dataset. Kernelized SVM classifiers and regression engines compute the squared distance between two sparse feature vectors by calculating the inner-product [1].

We propose an algorithm and an accelerator for SpMSpV, outperforming [1] by 70× while achieving a power efficiency gain of 8×. Our accelerator is specifically designed to be integrated into a CPU core. It shares the CPU cache and taps directly into the memory interface for faster access to the sparse matrix data.

This paper makes the following contributions:

- We present the SpMV algorithm and accelerator designed to be integrated into a CPU core. The proposed algorithm significantly reduces the memory access in sparse matrix by sparse vector multiplication, by fetching only the matrix elements that produce non-zero partial results;
- A fully associative dual port product cache is designed to further enhance the performance of the SpMSpV algorithm, by enabling an efficient accumulation of partial results.

The rest of this paper is organized as follows. Section 2 presents the background and motivation. Section 3 introduces the SpMSpV algorithm. Section 4 presents the sparse accelerator design. Section 5 discusses the evaluation results. Section 6 offers conclusions.

- *Leonid Yavits, E-mail: yavits@technion.ac.il.*
- *Ran Ginosar, E-mail: ran@ee.technion.ac.il.*

Authors are with the Department of Electrical Engineering, Technion-Israel Institute of Technology, Haifa 3200000, Israel.

Manuscript submitted: 05-Apr-2017. Manuscript accepted: 25-Apr-2017. Final manuscript received: 05-June-2017

2 Background and Motivation

A variety of dedicated hardware accelerators for sparse matrix multiplication have been proposed. Misra *et al.* [6] designed a parallel architecture comprising *nnz* processing elements (*nnz* is the number of nonzero matrix elements), and implemented a routing technique to improve the communication. Andersen *et al.* [2] suggested implementing sparse matrix multiplication on the Distributed Array Processor, a massively parallel SIMD architecture. Beaumont *et al.* [8] implemented matrix multiplication on a heterogeneous processing network. Wing [9] suggested a systolic array architecture, comprising a number of processing elements connected in a ring. Kieckhager *et al.* [12] used content addressable memory in the context of sparse matrix multiplication. Zhu *et al.* [10] suggested a 3-D stackable Logic-In-Memory (LiM) architecture. Yavits *et al.* [4] proposed an associative processor for sparse matrix multiplication. Some of the accelerator designs have been implemented in FPGA, for example by Zhuo *et al.* [5], Sun *et al.* [3] and Dorrance *et al.* [11].

We consider two ways of multiplying a matrix by a vector. The first is by implementing an inner dot product of each row of the matrix and the vector. In this paper, we refer to this method as CSR, because a sparse matrix is likely to be stored in CSR format if this method is employed. The second method is multiplying each column of the matrix by an appropriate vector element, and accumulating the product vector. We refer to this method as CSC, since a sparse matrix is likely to be stored in CSC format if such a method is employed.

Multiplying a sparse matrix by a dense vector (SpMDV) using either method has the computational complexity of $O(nnz_A)$ where nnz_A is the number of nonzero elements in the sparse matrix. The amount of data fetched from main memory (assuming the matrix is stored in main memory, while the vector is stored locally) is also $O(nnz_A)$. Hence, the arithmetic intensity of SpMDV (the ratio of performance to memory bandwidth) is O(1).

The situation is different when multiplying sparse matrix by sparse vector (SpMSpV). The approximate number

of nonzero pairs is $sps_B \times nnz_A$, where sps_B is the sparsity of the sparse vector (the ratio of the number of nonzero elements to the vector size). Hence the arithmetic intensity of SpMSpV implemented using the CSR method is O(sps_B). Assuming sparsity of 0.02% (as explained in Section 5), the performance of SpMSpV using CSR method is likely to be extremely low.

We aim at increasing the arithmetic intensity of SpMSpV using the CSC method (and thus increasing expected performance). We achieve that by reducing memory access by fetching only those matrix elements that end up producing nonzero pairs. Consequently, the arithmetic intensity of SpMSpV increases from O(sps_B) to O(1), similar to SpMV, leading to performance improvement by a similar factor. The proposed algorithm works equally well for both SpMSpV and SpMDV, as well as for sparse matrix by matrix multiplication.

3 SpMV CSC Algorithm

The SpMV algorithm is presented in Fig. 1. It multiplies a m×n sparse matrix A by a vector B of length n, to produce a result vector C of length m.

We compare our accelerator with the sparse accelerator introduced in [1]. The main advantage of our SpMV algorithm is the expected reduction of memory access by 3-4 orders of magnitude. This is made possible since our accelerator reads from memory only those nonzero elements of the sparse matrix A that are matched by a nonzero element of the sparse vector B. In contrast, reference [1] accelerator fetches from memory all nonzero elements of matrix A.

Algorithm 1 CSC SpMV

Let C(m) be a vector of size m, all zero initially.
Let A(m,n) be an m×n matrix of nnz_A nonzero elements, stored in CSC format, as follows:

- $A_{POINTER}$(n) is a double (8B) vector of column pointers;
- $A_{ROW\ INDEX}$(nnz_A) is a double vector (8B) of row indexes;
- A_{VALUE}(nnz_A) is a double vector (8B) of nonzero elements of A;
- Column *j* of A is held in

$A_{ROW\ INDEX}$ [$A_{POINTER\,J}$,..., $A_{POINTER\,J+1}$-1] (row indices) and A_{VALUE}[$A_{POINTER\,J}$,..., $A_{POINTER\,J+1}$-1] (values).
Let B be a column vector of nnz_B nonzero elements, as follows:

- B_{INDEX} is a double vector (8B) of indexes;
- B_{VALUE} is a double vector (8B) of nonzero elements of B;

Main:

```
1:  for (p = 0 ; p < nnzB; p++)     {
2:      j = BINDEX [p] ;
3:      bj = BVALUE [p] ;
4:      for (q = APOINTER [j] ; q < APOINTER [j+1] ; q++)    {
5:          i = AROW INDEX [q] ;
6:          aij = AVALUE [q] ;
7:          C [i] += aij * bj ;
8:      }
9:  }
```

Fig. 1: CSC SpMV algorithm

A significant disadvantage of this algorithm is that it accumulates the partial results (line 7 in Fig. 1) through all iterations. To implement such accumulation in a compressed CSC format, the following operations have to be performed: a product vector element index needs to be looked up in the product vector storage, the corresponding product vector element value needs to be fetched (to a multiplier-accumulator) and written back (after accumulation), and the product vector element index, if not found, needs to be stored alongside its value. Another option is to accumulate the product vector in a dense format. This eliminates the index lookup, but requires dense to CSC format conversion after completing the SpMV.

The CSC algorithm can be implemented in software, with no hardware acceleration. This is the baseline of our evaluation (Fig 5, speedup=1). However, the CSC algorithm can be significantly sped up by a specially designed hardware unit, an associative Product Cache, which supports the parallel implementation of those lookup, read and write operations. The Product Cache is described in Section 4.

Another disadvantage of the CSC algorithm is non-contiguous memory access (nonconsecutive matrix columns are typically fetched in CSC SpMSpV), which may reduce bandwidth and thus adversely affect the SpMSpV performance.

4 Accelerator Design

4.1 Architecture

The architecture of the SpMV accelerator is presented in Fig 2. The accelerator contains three main units: the sparse matrix Fetch Engine, the Floating point Multiplier-Accumulator (FMAC), and the Product Cache. A controller (FSM) that controls the operation of the accelerator and an internal memory buffer are not shown.

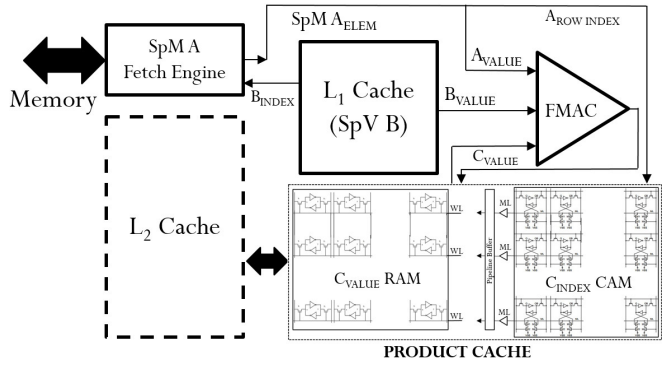

Fig 2. SpMV Accelerator Architecture, comprising sparse matrix Fetch Engine, FMAC and Product Cache.

Fig 2 also shows the cache and memory interfacing of our accelerator. Load and store commands similar to x86 MOVNTQ instruction are used to bypass the cache hierarchy while loading the sparse matrix data directly from memory. The resulting product vector can be stored directly into memory, or through the L_2 cache. The input vector B is assumed to be retained in the L_1 cache. Although each element of vector B is used only once in the CSC SpMV algorithm of Fig. 1, vector B is stored locally because its nonzero elements are needed in advance in order to index the relevant columns of matrix A in the memory, as

explained in Section 4.2.

The Product Cache comprises two juxtaposed memory arrays, one randomly accessed (RAM) for values, and another content addressable (CAM) for indices, where the word line (WL) of each RAM is connected (through a pipeline buffer) to the match line (ML) of the corresponding CAM row. This connection allows addressing the RAM content by a CAM row matched during a lookup. Both RAM and CAM are dual port memories, to enable simultaneous read and write in RAM, as well as simultaneous lookup and write in CAM, as explained in Section 4.2.

4.2 Operation

The accelerator dataflow is presented in Fig 3. The accelerator pipeline is presented in Fig 4. All data items are 8B wide (double precision). The operation is done in six steps.

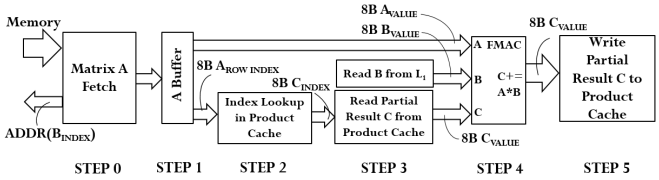

Fig 3. Accelerator Dataflow.

In step 0, the next sparse matrix A element A_{ELEM} (index and value) is prefetched from memory to an internal buffer. Step 0 is implemented in three phases, each taking several clock cycles. First, the relevant matrix A pointer is selected using the index of the next vector B nonzero element (line 2 of Fig. 1). Second, the address to the relevant matrix column is generated using the pointer (line 4 of Fig. 1). Third, the matrix element A_{ELEM} consisting of $A_{ROW\ INDEX}$ and A_{VALUE} is fetched from memory to the internal buffer (lines 5 and 6 of Fig. 1).

The rest of the steps are single-cycle (Fig 4). At step 1, $A_{ROW\ INDEX}$ and A_{VALUE} are read from the internal buffer. At step 2, the partial result index C_{INDEX} is looked up in the Product Cache using the row index $A_{ROW\ INDEX}$ of A_{ELEM} as the search key. The matching row of the CAM is tagged and further used to select the relevant row of the RAM array of the Product Cache, where the partial result value C_{VALUE} is stored. If there is no match, it means that such row index is used for the first time. At step 3, the partial product C_{VALUE} is read from the RAM to the FMAC (or reset, if there was no match). In parallel, the values of matrix A and vector B elements (A_{VALUE} and B_{VALUE}) are fetched to the FMAC from the memory buffer and L_1 cache respectively. At step 4, the new partial result (line 7 of Fig. 1) is calculated. At step 5, it is written back to the Product Cache (C_{VALUE} is written into the RAM, and the new C_{INDEX}= $A_{ROW\ INDEX}$ is added to the CAM if there was no match). The write-back of the k^{th} partial product coincides with the lookup for the $(k+3)^{th}$ $A_{ROW\ INDEX}$ and the read of the $(k+2)^{th}$ C_{VALUE}. To prevent pipeline stalls and associated delays, both RAM and CAM are made dual-ported.

After the SpMSpV is completed, the product vector C is stored in the Product Cache in a compressed format, in which the vector elements could be unordered (in contrast to a conventional compressed format, where the elements are arranged in an ascending index order). Hence, there are two options: The first option is to store the product vector to memory (directly or through the L_2 cache) in the unordered compressed format. The operation of our accelerator is not affected by the order of the vector or matrix elements. The second option is to reorder the product vector C in ascending index order before storing it to memory.

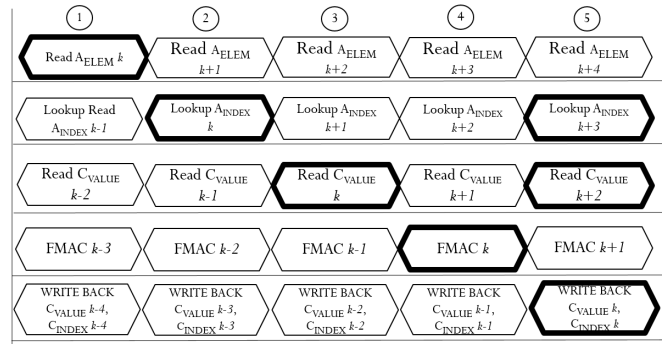

Fig 4. Accelerator Pipeline; bold lines at step 5 emphasize the concurrent read/write in the Product Cache.

4.3 Area and power

Assumptions are presented in Table 1. The estimated area of the SpMV accelerator is 0.14 mm^2. The power consumption of the SpMV accelerator is dominated by the CAM array in the Product Cache, since it is constantly active (one compare and one write every cycle) during the accelerator operation. The overall estimated power consumption of the SpMV accelerator is 0.4W.

TABLE 1. ACCELERATOR AREA AND POWER ASSUMPTIONS

Parameter	Value
Product Cache Size	4k cache lines
RAM width	64b
CAM width	64b
RAM bitcell size, 14nm	$0.06\mu^2$ [13]
CAM/RAM cell size ratio	1.5
Overhead	40%
FMAC area	$0.0625mm^2$ [1]
CAM cell energy, 14nm	1fJ
Operating Frequency	750MHz

5 EVALUATION

5.1 Methodology

To evaluate the performance of the SpMV accelerator, we use 900 real-valued sparse matrices from the UFL Sparse Matrix Collection [13] with the number of nonzero elements spanning 100k to 760m. The input vector B is a randomly selected row of matrix A. The median sparsity of UFL matrices is 0.0002 (0.02%). To enhance the validity of this evaluation, we repeat the same matrix run 100 times, each time with a different (randomly selected row) sparse vector.

5.2 Simulation results

We simulate the SpMSpV using a cycle accurate simulator of our accelerator. We compare our accelerator with the reference [1] sparse accelerator and with an un-accelerated execution of CSC SpMSpV algorithm of Fig. 1 on Intel i7-6600U CPU with 16GB DRAM, running at 2.6GHz.

Fig 5 presents the simulated speedup of SpMSpV implemented on our accelerator (assuming the product vector is reordered into the conventional compressed format), as well as the speedup of reference accelerator [1] over the unaccelerated CPU implementation, as function of nnz_A (a) and B vector sparsity sps_B (b).

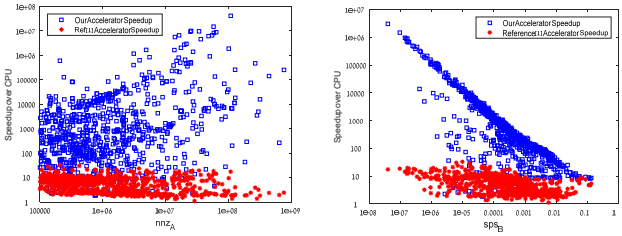

Fig 5. Simulated Speedup of SpMSpV vs. (a) nnz_A, (b) sps_B.

The median simulated speedup of our accelerator for all 900 test matrices is 257×. The median speedup of our accelerator relative to the reference [1] accelerator is 70×. The speedup tends to increase with growing number of nonzero elements of matrix A, but decreases with B vector sparsity.

The efficiency of our accelerator declines as input matrix and vector become denser. For SpMDV, the performance of our accelerator is limited by the number of FMACs. If it is increased to four, the SpMDV performance of our accelerator will match that of reference [1] accelerator. In such adaptive design, three out of four FMACs are operational in SpMDV but shut down during SpMSpV.

Fig 6(a) (left y-axis) presents the cumulative distribution of the number of nonzero elements in the product vector C. 91.6% of all SpMSpV runs produce vector C of fewer than 4k nonzero elements. Hence our design choice of the Product Cache size (4k). If nnz_C exceeds 4k, some of the partial results can be evicted from the Product Cache and replaced by new results using one of the conventional cache replacement mechanisms, for example LRU. The effect of such eviction on the overall execution time is negligible. The median speedup of our accelerator as a function of product cache size is also shown in Fig 6(a) (right y-axis).

The median power efficiency of our accelerator (calculated as simulated speedup over power) relative to reference [1] accelerator is 8× (assuming the product vector is sorted into the conventional compressed format). Overall, we estimate approximately 29× average energy reduction for the SpMSpV based applications (kernelized SVM classification K-SVM-C, kernelized SVM regression K-SVM-C and sparse PCA). The energy reduction figures are shown in Fig 6(b), compared to reference [1] accelerator.

6 Conclusions

We propose an algorithm and an accelerator for Sparse Matrix by Vector Multiplication (SpMV), which is an essential kernel in machine learning and data mining applications. Our accelerator is specifically designed to be integrated into a CPU core. It shares the CPU cache and taps directly into memory interface for faster access to the sparse matrix data.

Our accelerator is particularly efficient in accelerating of sparse matrix by spare vector multiplication, since it reduces memory access by 3-4 orders of magnitude. It is shown to outperform an existing in-CPU accelerator by 70x while achieving 8x better power efficiency, resulting in 29x overall energy reduction for the SpMSpV based applications.

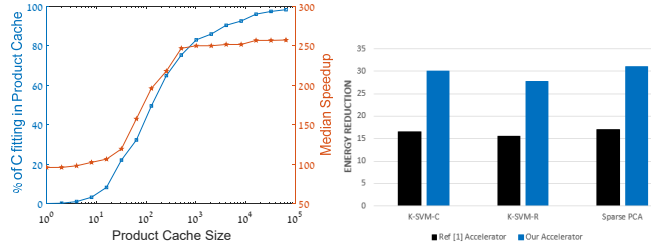

Fig 6. (a) Cumulative Distribution of nnz_C (left) and Median Speedup (right) vs. Product Cache Size (b) Normalized Energy Reduction for SpMSpV Based Kernels, our vs. ref [1] accelerator.

References

[1] A. Mishra, Nurvitadhi, E., Venkatesh, G., Pearce, J., and Marr, D, "Fine-grained accelerators for sparse machine learning workloads," IEEE ASP-DAC, pp. 635-640, 2017

[2] J. Andersen, G. Mitra, D. Parkinson. "The scheduling of sparse matrix-vector multiplication on a massively parallel DAP computer." Parallel Computing 18, no. 6 (1992): 675-697.

[3] J. Sun, G. Peterson, O. Storaasli. "Sparse matrix-vector multiplication design on FPGAs." IEEE Symposium on FCCM, pp. 349-352, 2007.

[4] L. Yavits, Morad, A., and Ginosar, R. (2015)," Sparse matrix multiplication on an associative processor." IEEE Transactions on Parallel and Distributed Systems, 26(11), 3175-3183.

[5] L. Zhuo, V. Prasanna. "Sparse matrix-vector multiplication on FPGAs." 13th international symposium on FPGA, pp. 63-74. ACM, 2005.

[6] M. Misra, D. Nassimi, V. Prasanna. "Efficient VLSI implementation of iterative solutions to sparse linear systems." Parallel Computing 19, no. 5 (1993): 525-544.

[7] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," In Proc. High Performance Computing Networking, Storage and Analysis, pp 18:1-18:11, 2009.

[8] O. Beaumont, Boudet, V., Rastello, F., and Robert, Y. (2001). Matrix multiplication on heterogeneous platforms. IEEE Transactions on Parallel and Distributed Systems, 12(10), 1033-1051.

[9] O. Wing, "A content-addressable systolic array for sparse matrix computation." Journal of Parallel and Distributed Computing 2, no. 2 (1985): 170-181.

[10] Q. Zhu, Graf, T., Sumbul, H. E., Pileggi, L., and Franchetti, F. "Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware," In IEEE HPEC 2013 (pp. 1-6)

[11] R. Dorrance, Ren, F., and Marković, D. "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-BLAS on FPGAs", 2014 international symposium on FPGA.

[12] R. Kieckhager, C. Pottle, "A processor array for factorization of unstructured sparse networks", IEEE CCC, 1982, pp. 380-383.

[13] S. Natarajan, et al. "A 14nm logic technology featuring 2nd generation FinFET, air-gapped interconnects, self-aligned double patterning and a 0.0588 µm² SRAM cell size," IEDM, 2014.

[14] T. Davis, Y. Hu, "The University of Florida sparse matrix collection," ACM Transactions on Mathematical Software, 38, no. 1 (2011).