SPARSE MATRIX MULTIPLICATION ON AN ASSOCIATIVE PROCESSOR

L. Yavits, A. Morad, R. Ginosar

Abstract—Sparse matrix multiplication is an important component of linear algebra computations. Implementing sparse matrix multiplication on an associative processor (AP) enables high level of parallelism, where a row of one matrix is multiplied in parallel with the entire second matrix, and where the AP execution time of vector dot product does not depend on the vector size. Four sparse matrix multiplication algorithms are explored in this paper, combining AP and CPU processing to various levels. They are evaluated by simulation on a large set of sparse matrices. The computational complexity of sparse matrix multiplication on AP is shown to be an *O(M)* where M is the number of nonzero elements. The AP is found to be especially efficient in binary sparse matrix multiplication. AP outperforms conventional solutions in power efficiency.

Index Terms— Sparse Linear Algebra, SIMD, Associative Processor, Memory Intensive Computing, In-Memory Computing.

1 Introduction

Sparse matrix multiplication is a frequent bottleneck in large scale linear algebra applications, especially in data mining and machine learning [28]. The efficiency of sparse matrix multiplication becomes even more relevant with the emergence of big data, giving rise to very large vector and matrix sizes.

Associative Processor (AP) is a massively parallel SIMD array processor [15][22][43]. The AP comprises a modified Content Addressable Memory (CAM) and facilitates processing in addition to storage. The execution time of a typical vector operation in an AP does not depend on the vector size, thus allowing efficient parallel processing of very large vectors. AP's efficiency grows with the data set sizes and data-level parallelism. A detailed description of the AP architecture, functionality and associative computing can be found in [23].

Associative processing has been known and extensively studied since the 1960s. Commercial associative processing never quite took off, because only limited amounts of memory could be placed on a single die [21]. Equally important, standalone bit- and word-parallel conventional SIMD processors outperformed APs due to the data sets and tasks of limited size. However, the progress in computer industry and semiconductor technology in recent years opens the door for reconsidering the APs:

- The rise of big data pushes the computational requirements to levels never seen before. The amounts of data to be processed simultaneously require a new parallel computing paradigm. Unlike conventional SIMD processors, the performance and efficiency of an AP improves with the data set size.
- Power consumption, which used to be a secondary factor in the past, has become a principal constraint on scalability and performance of the parallel architectures. The AP is shown to achieve a better power efficiency [23].
- Off-chip memory bandwidth is another factor limiting the performance and scalability of parallel architectures. Associative processing mitigates this limitation by intertwining computing with data storage.
- In high performance dies, thermal density is becoming the limit on total computation capabilities; associative processing leads to uniform power and thermal distribution over the chip area, avoiding hot spots and enabling the three dimensional (3D) integration.

In this work, we present four different algorithms of sparse matrix-matrix multiplication on the AP. The first algorithm, designated "AP", is a fully associative implementation, making use only of the intrinsic AP resources. We show that the computational complexity of a fully associative implementation is $O(M)$, where M is the number of nonzero elements. In the second algorithm, called "AP+ACC", the singleton products are computed by the AP and an external CPU is used to accumulate them. The third algorithm, "AP+MULT", uses a CPU to multiply matrix elements; the products are accumulated by the AP. The fourth algorithm, "AP+MULT+ACC", uses the AP for matching the matrix elements, and a CPU for both multiplication and accumulation. We find that the fully associative implementation is especially efficient for very large matrices with high number of nonzero elements per row. Fully associative implementation is also preferred for multiplication of binary sparse matrices (that is, matrices where the nonzero elements are ±1). In contrast, the other three (hybrid) algorithms are more efficient for matrices with a lower number of nonzero elements per row, and their efficiency improves slower or remains constant with the number of nonzero elements.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 presents associative algorithms for sparse matrix multiplication. Section 4 de-

- *Leonid Yavits (*), E-mail: yavits@tx.technion.ac.il.*
- *Amir Morad (*), E-mail: amirm@tx.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 32000, Israel.

tails the evaluation methodology and presents the simulation results. Section 5 offers conclusions.

2 RELATED WORK

While this paper studies a sparse matrix-matrix multiplication, a majority of previous studies have targeted sparse matrix-vector multiplication (SpMV). For simplicity, in this section we apply the term "sparse matrix multiplication" (SpMM) to both problems.

A substantial body of literature explores sparse matrix multiplication optimization techniques. A comprehensive review of these techniques is provided by R. Vuduc [36]. We take a slightly different look, focusing on hardware platforms rather than on software implementation. The literature can be divided into three categories, as summarized in TABLE 1.

TABLE 1
RELATED WORK SUMMARY

Category	Existing Work
General Purpose Computers	Off-the-shelf [1][7][39][44] Advanced multicore [40] Manycore supercomputer [5]
GPU	[9][17][26][28][29][38]
Dedicated Hardware Solutions	FPGA [20][25] Manycore Processor [27] Distributed Array Processor [16] Systolic Processor [32] Coherent Processor [4] TCAM / PIM [11] Heterogeneous platform[30][31] 3D LiM [33]

The first category targets the optimization of sparse matrix multiplication on general purpose computer architectures. S. Toledo [39] enhanced sparse matrix multiplication on a superscalar RISC processor by improving instruction-level parallelism and reducing cache miss rate. A. Pinar *et al.* [1] proposed further optimization of data structures using reordering algorithms, to improve cache performance. E. Im *et al.* [7] developed the SPARSITY toolkit for the automatic optimization of sparse matrix multiplication. Y. Saad *et al.* [44] proposed PSPARSLIB, a collection of sparse matrix multiplication subroutines for multiprocessors. S. Williams *et al.* [40] examined and optimized sparse matrix multiplication across a broad spectrum of multicore architectures. Finally, Bowler *et al.* [5] optimized sparse matrix multiplication for a 512-core supercomputer.

Another direction is the implementation and optimization of sparse matrix multiplication using GPU. While this effort still relies on a conventional computational platform and focuses mainly on algorithm optimization, it enables significant speedup over sequential CPU or even multicore solutions [29]. Many of the GPU-based studies rely on G. Blelloch's [9] research into mapping of sparse data structures onto SIMD hardware. S. Sengupta *et al.* [38] developed segmented scan primitive for efficient sparse matrix multiplication on GPU. J. Bolz *et al.* [17] implemented a sparse matrix solver on GPU. M. Baskaran *et al.* [26] enhanced GPU sparse matrix multiplication by creating an optimized storage format. Bell *et al.* [28][29] develop methods to exploit common forms of matrix structure while offering alternatives to accommodate irregularity.

The third direction encompasses special purpose hardware solutions for sparse matrix multiplication. L. Zhuo [25] proposed an FPGA based design, which reportedly demonstrated a significant speedup over then-current general-purpose solutions (such as Itanium 2), especially for matrices with very irregular sparsity structures. Another FPGA based sparse matrix multiplication solution was introduced by J. Sun *et al.* [20]. Some specialty solutions relying on VLSI implementation have been suggested as well. M. Misra *et al.* [27] developed a parallel architecture comprising M processing elements (where M is the number of nonzero elements in a matrix), and implemented an efficient routing technique to resolve the communication bottleneck. J. Andersen *et al.* [16] suggested implementing sparse matrix multiplication on the Distributed Array Processor (DAP), a massively parallel SIMD architecture. O. Beaumont *et al.* [30][31] implemented matrix multiplication on a heterogeneous network.

A number of hardware solutions using content-addressable memory have also been proposed. O. Wing [32] suggested a systolic array architecture, comprising a number of processing elements connected in a ring. Each processing element has its own content-addressable memory, storing the nonzero elements of the sparse matrix. Matrix elements are extracted from the memory by content addressing. Sparse matrix-vector multiplication takes $O(M)$ cycles (where M is the number of nonzero elements in matrix). That work relies on an earlier study by R. Kieckhager *et al.* [35], who were probably the first to use a content-addressable memory in the context of sparse matrix multiplication. Q. Guo *et al.* [11] implemented a fixed point matrix multiplication on a TCAM based Processing-In-Memory (PIM) architecture. They use TCAM to match key-value pairs but rely on a microcontroller for multiplication. Recently, Q. Zhu *et al.* [33] suggested a 3-D Logic-In-Memory (LiM) architecture where DRAM dies are intertwined with logic dies in a 3D stack. Their architecture uses a logic-enhanced CAM to take advantage of its parallel matching capabilities.

Associative processors have also been considered in the context of matrix processing. C. Stormon [4] introduced the Coherent Processor, a massively parallel associative computer. Sparse matrix computations are mentioned among the Coherent Processor's applications although no details of the sparse matrix multiplication are provided. Stromon suggested using the Coordinate (COO) format of storing nonzero elements of sparse matrices along with their row and column indices, in contrast other sparse formats such as Compressed Sparse Row (CSR) or ELLPACK (ELL) [18], which are more efficient for sequential processors or GPUs.

The key contribution of the present work is the efficient implementation of sparse matrix multiplication on a memory intensive associative processor (AP), verified by extensive AP simulation using a large collection of sparse matrices [41].

3 Sparse Matrix Multiplication on AP

In this section we detail the sparse matrix multiplication algorithm and its four implementations on the AP.

Fig. 1 illustrates the multiplication of sparse matrix A by sparse matrix B. In this example, row j of matrix A has three nonzero elements in columns $\{i_1,\ i_2,\ i_3\}$. Rows $\{i_1, i_2, i_3\}$ of matrix B have nonzero elements in columns $\{k_1, k_3, k_5\}$, $\{k_2, k_4\}$ and $\{k_1, k_2, k_5\}$, respectively.

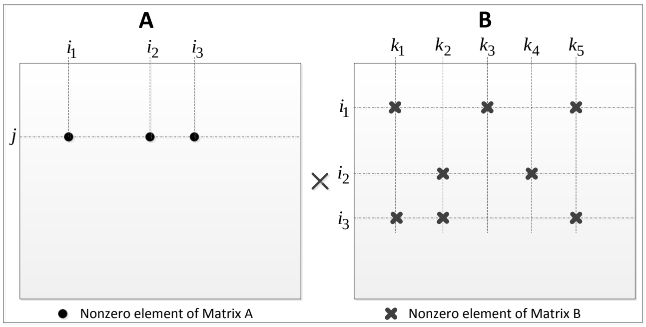

Fig. 1. Sparse Matrix Multiplication - Illustration

Fig. 2 shows the associative processing array and reduction tree [23] mapping. We assume that both input matrices are stored in the AP in the COO format, where nonzero elements are entered consecutively, with the row and column indexes stored alongside the matrix element.

An AP implementation does not require the matrix elements to be stored in any particular order. Hence the Matrix Market (MM) [34] sparse format is supported as well.

Fig. 3 presents the pseudo code of the fully associative sparse matrix multiplication (algorithm "AP"). It includes two internal loops nested within an external one. The external loop goes over the nonzero rows of matrix A. The first internal loop goes over the nonzero elements in each nonzero row of matrix A and takes three steps. At step 1, a nonzero element of row j and its column index i are read from the associative memory (associative processing array). At step 2, its column index i is compared against the row index field of the entire matrix B. This step is done in parallel for all nonzero elements of matrix B, using the AP compare command. All matching nonzero elements of matrix B (k_1, k_3 and k_5 for row i_1 etc. in Fig. 1 and Fig. 2) are tagged. At step 3, the nonzero element of matrix A is written simultaneously into all tagged rows, alongside the tagged elements of matrix B (segments i_1, i_2 and i_3 of Fig. 2).

The first internal loop is repeated while there are nonzero elements in row j of matrix A. Upon completion, all nonzero pairs of matrices A and B required to calculate the row j of the product matrix C are aligned (stored in the same associative processing unit) in the associative processing array.

Next step 4 is the associative multiplication of A,B pairs, performed in parallel for all pairs. For instance, the index of the first product in Fig. 2 is j, i_1, k_1.

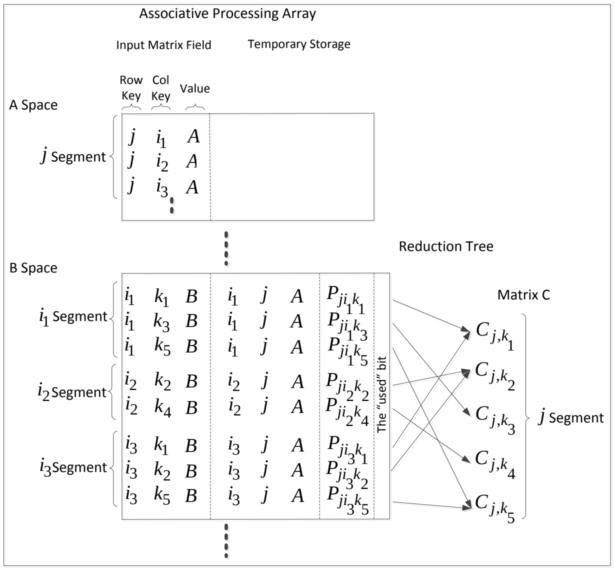

Fig. 2. AP Memory and Reduction Map

```
Init      {
              Matrix A → A space;
              Matrix B → B space;
}

Main {
    While (!end of A) {  //serially over all nz rows of A
        While (!end of row j) { //serially, over all nz elements in j^th row of A
            1.  Read_next {i, A_j,i}
            2.  Tag all B_i,k   //in parallel, single step, for all k
            3.  Write A_j,i    //in parallel, single step, into all tagged rows
        }
        4.      P_j,i,k = ASSOCIATIVE_MULT(A_j,i, B_i,k) //forall aligned pairs

        While (∃k not used)  {  //serially over all values k
            5.  Read_next {k, P_j,i,k}  //find next not used k value
            6.  Tag all P_j,i,k   //parallel forall P_j,*,k with same k, single step
            7.  Mark "used" //parallel forall tagged rows, single step
            8.  C_j,k = ASSOCIATIVE_REDUCE_SUM(P_j,i,k)
        }
}
```

Fig. 3. AP algorithm for fully associative sparse matrix multiplication

The second loop sums up the products (the singletons). It contains steps 5 through 8. At step 5, a singleton product is read from the associative processing array (beginning with the first one). At step 6, its B column index k (unless it is marked "used") is compared against the B column index of all singleton products, and all singletons with B column index k are tagged. At step 7, the tagged rows are marked "used" by a write command. Those tagged rows hold the singleton products that need to be accumulated to form element $C_{j,k}$. Step 8 is the reduction. The reduction tree is pipelined hence the loop may end without waiting for the reduction tree to complete. The loop is repeated while there are unprocessed (that is, not marked "used") B column indices.

In certain sparse matrices, most rows and columns contain very few nonzero elements. In such cases, parallel reduction (step 8 in Fig. 3) may be less efficient because a very few singleton products are accumulated in each iteration. Consequently, the reduction may better be carried out word-serially, by an external CPU. That algorithm, "AP+ACC," is shown in Fig. 4. Steps 1 through 6 are identical to those of "AP". The 8th step is a nested loop that goes over all the singleton products tagged at step 6. Each $P_{j,i,k}$ singleton is read and accumulated by an external CPU. We assume a pipelined operation so that steps 8a and 8b in Fig. 4 are performed in parallel; once the pipeline is filled, each pass of the loop takes a single cycle.

Same code as in Fig. 3, except:

8. Forall tagged rows // serially
    a. Read $P_{j,i,k}$
    b. $C_{j,k}$=CPU_ACC ($C_{j,k}, P_{j,i,k}$)

Fig. 4. "AP+ACC" algorithm, using serial accumulation

Similarly, a parallel associative multiplication (step 4 in Fig. 3) may be inefficient when the average number of nonzero elements per matrix row is small. In such case, the multiplication of matrix elements may be best performed word-serially by an external CPU. Fig. 5 presents the pseudo code of this "AP+MULT" algorithm. Steps 1, 2 and 5 through 8 are identical to those of "AP". The 3rd step is a nested loop that goes over all the elements of matrix B with the row index matching the column index i of the nonzero element $A_{j,i}$. Each $B_{i,k}$ element is multiplied by $A_{j,i}$ at the external CPU and is written back to the corresponding row of the associative processing array. We assume a pipelined operation so that steps 3b and 3c in Fig. 5 are performed in parallel; once the pipeline is filled, each pass of the loop takes 2 cycles.

Same as Fig. 3, except:

3. Forall $B_{i,k}$ // serially
    a. Read_next $B_{i,k}$; // single step
    b. $P_{i,k}$=CPU_MULT ($A_{j,i}, B_{i,k}$)
    c. Write $P_{i,k}$ alongside $B_{i,k}$; // single step

line 4 is deleted

Fig. 5. "AP+MULT" algorithm using serial multiplication

Both algorithms "AP+MULT" and "AP+ACC" are combined into "AP+MULT+ACC" in Fig. 6. This algorithm is efficient for smaller matrices with a lower average number of nonzero elements per row (for example, diagonal matrices).

4 Simulations of SpMM on AP

The AP simulator [23] is used to quantify the efficiency of the four algorithms of Section 3. The experimental setup, matrix statistics and simulation results are described in this section.

4.1 Experimental Setup

To simulate sparse matrix multiplication, we use 900 square matrices with the number of nonzero elements spanning from ten thousand to eight million, randomly selected from the collection of sparse matrices from the University of Florida [41].

Init {
  Matrix A → A space;
  Matrix B → B space;
}

Main {
  While (!end of A) { //serially over all nz rows of A
    While (!end of row j) { //serially, over all nz elements in j^{th} row of A
      1. Read_next $\{i, A_{j,i}\}$
      2. Tag all $B_{i,k}$ //in parallel, single step, for all k
      3. Forall $B_{i,k}$ // serially
        a. Read_next $B_{i,k}$; // single step
        b. $P_{i,k}$=CPU_MULT ($A_{j,i}, B_{i,k}$)
        c. Write $P_{i,k}$ alongside $B_{i,k}$; // single step
    }
    4. $P_{j,i,k}$ = ASSOCIATIVE_MULT($A_{j,i}, B_{i,k}$) //forall aligned pairs

    While (∃k not used) { //serially over all values k
      5. Read_next $\{k, P_{j,i,k}\}$ //find next not used k value
      6. Tag all $P_{j,i,k}$ //parallel forall $P_{j,*,k}$ with same k, single step
      7. Mark "used" //parallel forall tagged rows, single step
      8. Forall tagged rows // serially
        a. Read $P_{j,i,k}$
        b. $C_{j,k}$=CPU_ACC ($C_{j,k}, P_{j,i,k}$)
    }
}

Fig. 6. "AP+MULT+ACC" algorithm using serial multiplication and accumulation

In our simulation, we assume that the entire workload fits in the internal memory of the AP. This assumption is reasonable for the matrices of these sizes. The assumption of the data being resident in a device memory is quite custom in SpVM and SpMM performance sis [19][29]. Larger matrices would have to be partitioned for external multiplication on AP.

We simulate the sparse matrix multiplication using the AP simulator [23]. As shown in Fig. 2, each pair of matrix elements and the resulting singleton product are processed by a single AP processing unit. Simulations are performed on Intel® Core™ i7-3820 CPU with 32GB RAM, and simulation times for the 10K—8M nonzero element matrices range between few tens of seconds and few tens of hours.

4.2 Matrix Statistics

AP performance depends on the data wordlength rather than on data set size [23].

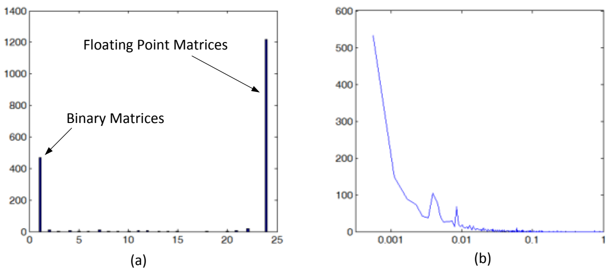

Fig. 7. (a) Wordlength histogram, (b) Histogram of the average number of nonzero elements per row, relatively to the matrix dimension

Fig. 7(a) presents the matrix element wordlength histogram. The wordlength is implied by analysis of the matrix elements (which are originally available from the University of Florida collection in MATLAB format). The first peak represents the binary matrices (two bits stand for a value bit and a sign). The second peak encapsulates matrices with floating point data elements (24 bits IEEE754 single precision mantissa). In this work, all multiplications are carried out as either binary (Boolean) or floating point operations.

There are several applications that use sparse binary matrices. According to [41], these applications may include recommender systems, undirected graph sequencing, certain optimization problems, duplicate structural problems, random un-weighted graph processing and computational fluid dynamics problems. To emphasize the efficiency of the "AP" algorithm, we employ parallel Boolean multiplication in the binary matrices: it takes only eight cycles, regardless of the number of nonzero elements in a row.

As we show in Section 4.3 below, the performance of the fully associative "AP" algorithm is strongly affected by the average number of nonzero elements per row. The distribution of the average number of nonzero elements per row relative to the matrix dimension is shown in Fig. 7(b).

In "AP" and "AP+ACC" algorithms, we calculate the singleton products by associatively multiplying the matrix elements. Consider a matrix containing a limited number of unique elements, known in advance. In such case, the products of all unique elements can be precalculated, and a "vocabulary" containing all pairs of the unique elements and their products can be created. Instead of multiplication, the AP would then match the pairs of the unique elements and substitute the precalculated product in the result field. For n unique elements in a matrix, such vocabulary-based multiplication would take $2n^2$ cycles. Hence, if $2n^2$ is shorter than the associative multiplication time (in cycles), the "AP" and "AP+ACC" algorithms can be sped up by replacing associative multiplication by vocabulary-based one.

Fig. 8 shows the distribution of the $2n^2$ figure. The first peak corresponds to binary matrices and should therefore be excluded from the analysis. All values to the left of the 8,800 (the floating point associative multiplication cycle count) mark on the horizontal axis belong to the group for which vocabulary multiplication is preferred. For the rest of the $2n^2$ values, the number of the unique elements n is too large for the vocabulary multiplication to be time-efficient. The percentage of matrices with the number of unique elements in the left field (excluding binary matrices) is around 15%.

We do not implement the vocabulary multiplication in our simulations, but find it worth noticing as an additional potential benefit of associative processing as compared to a conventional (CPU or GPU) multiplication.

For statistical analysis, we examined approximately 1700 square matrices of different sparsity structures, dimensions and nonzero element counts to receive a statistically significant outcome.

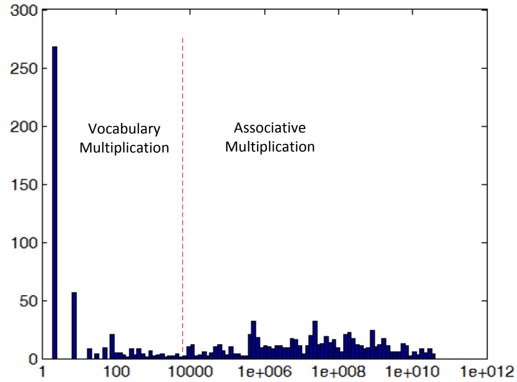

Fig. 8. $2n^2$ histogram, showing number of matrices having n unique elements

4.3 Simulation Results

Fig. 9 presents the SpMM execution time of the four algorithms of Section 3 for the matrices with floating point elements (a) and with binary elements (b).

The reason for the spread in execution time (per each number of nonzero elements) in each individual algorithm is the sensitivity of the associative implementation to the number of nonzero rows and average number of nonzero elements per row. For two matrices with a similar number of nonzero elements, two orders of magnitude difference in the average number of nonzero elements per row cause a similar difference in the execution time. For example, the "Williams/webbase-1M" matrix has 3,105,536 nonzero elements and an average of 3.1 nonzero elements per row. The "ND/nd3k" matrix however has 3,279,690 nonzero elements but an average of 364.4 nonzero elements per row. The multiplication of each of those two matrices by itself using the "AP" algorithm takes 8.7 and 0.17 billion cycles respectively, a difference of almost two orders of magnitude.

This sensitivity of performance to the average number of nonzero elements per row is shared, although possibly to a lesser extent, by conventional SpMV and SpMM implementations (on CPU and GPU) [19][42].

The difference in execution times of the "AP" algorithm with respect to binary vs. floating point matrices is a result of the difference in Boolean vs. associative multiplication times.

For smaller matrices (having less than one million nonzero elements), the "AP+MULT+ACC" algorithm provides the best performance in most cases, with the exception of binary matrices. For binary matrices, the picture is mixed. Even for the smallest matrices, the "AP" sometimes outperforms the hybrid algorithms, due to time-efficient Boolean multiplication.

As the number of nonzero elements approaches one million, the performance of the "AP" algorithm gradually improves. For matrices of several millions of nonzero elements, "AP" tends to outperform the hybrid algorithms.

MATLAB's nonlinear least square solver ***lsqcurvefit*** has been used to estimate the computational complexity of the associative SpMM algorithms. The result of the Least Square Error (LSE) interpolation is shown in Fig. 9(a) and (b), implying that the computational complexity of associative SpMM is $O(M)$, where M is the number of nonzero elements.

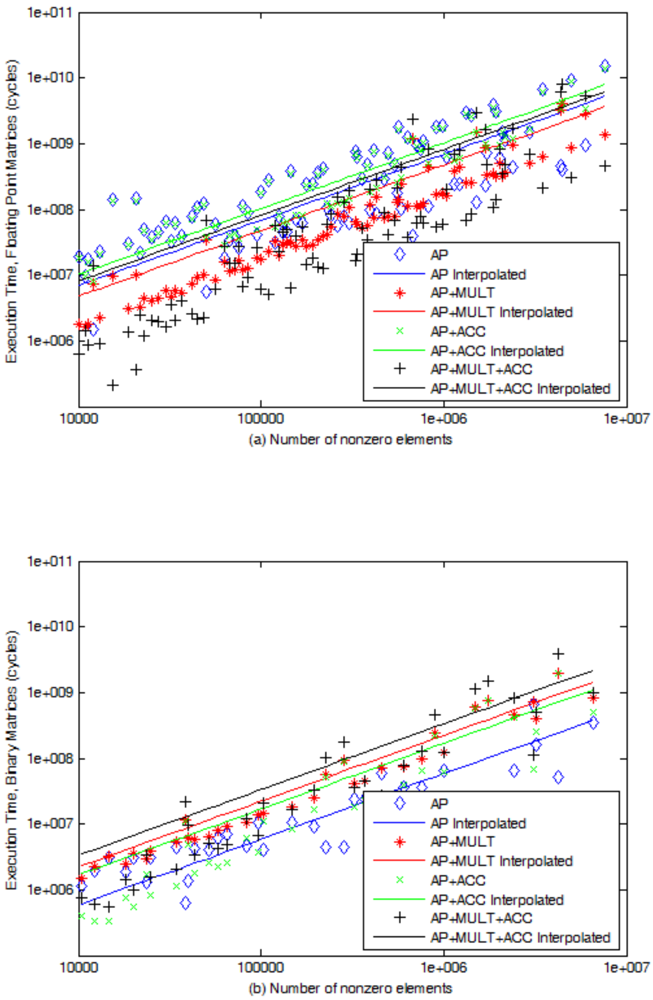

Fig. 9. Execution time vs. number of nonzero elements: (a) Floating point matrices; (b) Binary matrices

The performance of the "AP" algorithm for floating point and binary matrices as functions of the number of nonzero elements, as well as the LSE-interpolated performance of the hybrid algorithms are presented in Fig. 10 and Fig. 11.

For comparison, Fig. 10 and Fig. 11 also show the performance of Nehalem and NVidia GTX285 based solutions [19], the performance of NVidia GTX280 over structured and unstructured matrix SpMV [29], as well as the performance of an FPGA based solution [25]. The operating frequency of the AP is assumed to be 3GHz.

The spread in "AP" performance is a function of the average number of nonzero elements per matrix row. The divergence between binary and floating point performance is a result of Boolean vs. associative multiplication time difference.

The difference in performance of the "AP" sparse algorithm relative to the CPU and GPU based solutions is a result of a relative inefficiency of associative arithmetic when applied in parallel to small sets of numbers. An associative multiplication in the "AP" algorithm is performed once per matrix row.

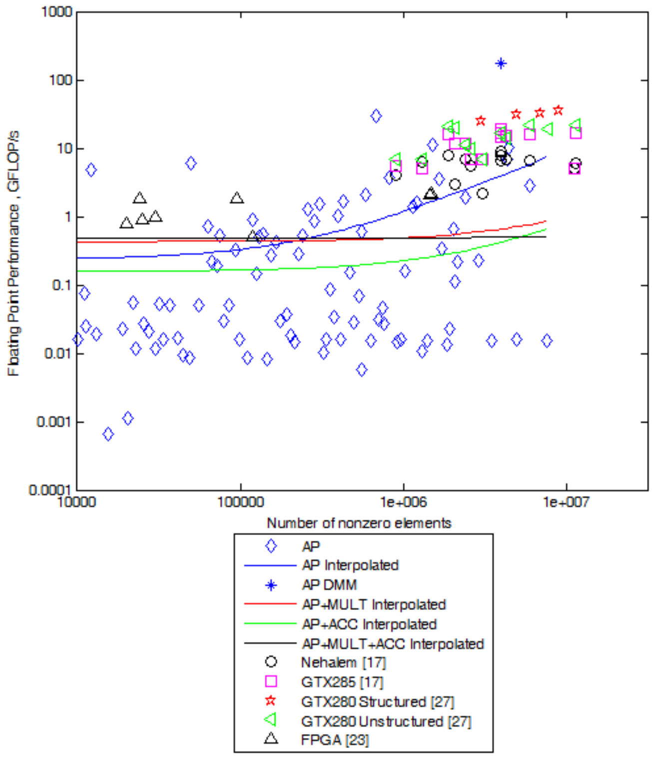

Fig. 10. Floating point matrix performance vs. number of nonzero elements

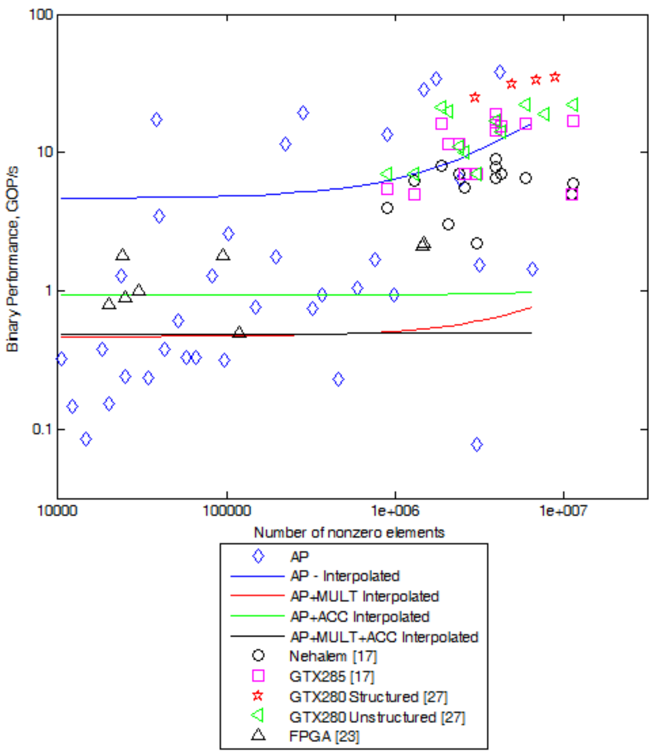

Fig. 11. Binary matrix performance vs. number of nonzero elements

If the average number of nonzero elements per row is small (which is consistently the case in University of Florida collection matrices), the effectiveness of the "AP" algorithm is limited. "AP" is least efficient for diagonal matrices, where there is only one multiplication per nonzero row. On the other end of the efficiency scale is dense matrix multiplication, where an associative multiplication is applied to N^2 matrix elements in parallel (N is the matrix dimension) per each matrix row. For comparison, a 2000×2000 dense matrix multiplication (DMM) performance is also shown in Fig. 10.

As the number of nonzero elements per row grows, the efficiency of associative arithmetic increases. This is illustrated by the curving upwards of the LSE-interpolated AP performance charts in Fig. 10 and Fig. 11. As expected, the performance of the hybrid algorithms grow much slower or remains constant.

The sparsity structure of a matrix seems to have little effect on the associative implementation. This stands in contrast with the GPU implementations which seem to perform better when multiplying structured matrices [29].

Fig. 12 presents the power efficiency (performance to power ratio) of the "AP" algorithm for floating point (a) and binary (b) matrices, as functions of the number of nonzero elements. For comparison, Fig. 12(a) and (b) also show the power efficiency of NVidia GTX285 [19] and NVidia GTX280 [29] based SpMV, where we use the "graphic part only" power figures as published in GTX280 and GTX285 data sheets [13][14]. The power of FPGA based solution [25] was not reported. The average SpMM power consumption of the AP is sub 2W since only a very small fraction of the AP processing units is active at a time. This AP power efficiency advantage stems from in-memory computing (there are no data transfers between processing units and memory hierarchies) and from low-power design made possible by the very small size of each processing unit. The power efficiency of the DMM by "AP" is shown in Fig. 12(a) as well.

A noticeable limitation of the "AP" algorithm is the sequential processing of the matrix rows (the outer loop of Fig. 3). A parallelization of matrix row processing may significantly improve the performance of the "AP" algorithm. For example, diagonal matrices can easily be processed in a row-parallel manner, since there is only one nonzero singleton product per each matrix row. An optimization of the "AP" algorithm is the subject of our future work.

5 Conclusions

Sparse matrix multiplication is of great importance for many linear algebra applications, especially machine learning. The efficient implementation of sparse matrix multiplication becomes even more critical when applied to big data problems.

An Associate Processor (AP) is essentially a large memory with massively-parallel processing capabilities. It offers dual use: either a CPU accesses the data in that memory, or the data is being processed associatively within the same memory. This paper investigates the merit of implementing sparse matrix multiplication on the AP.

We propose and compare four algorithms for the AP, from a fully associative computation to a hybrid of AP and CPU. To quantify the efficiency of the proposed algorithms, we simulate them using a large variety of sparse matrices.

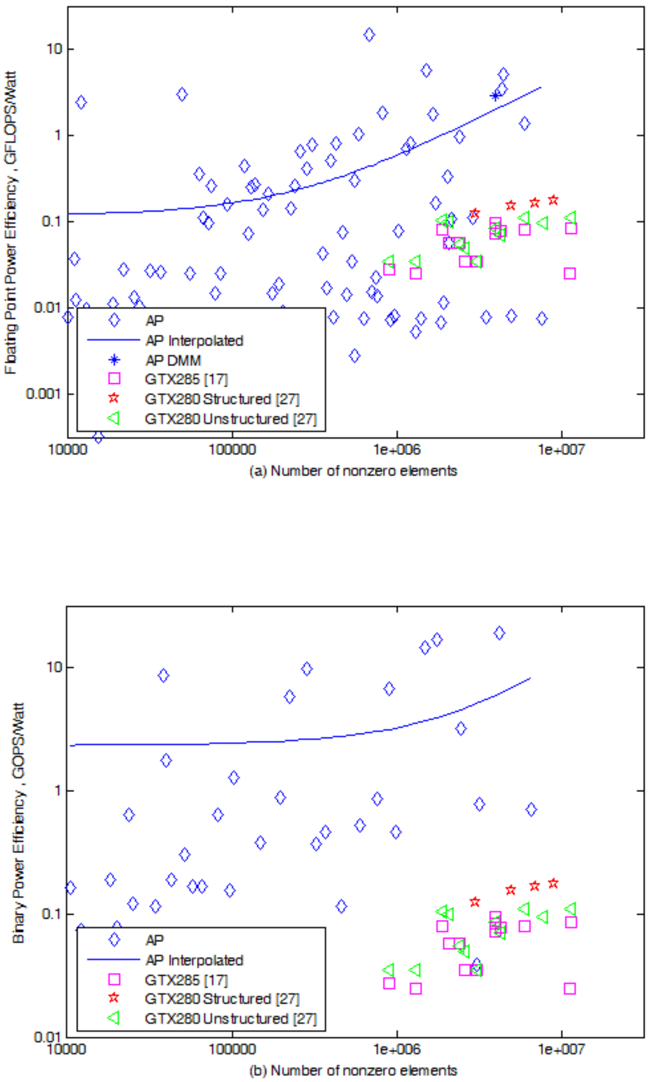

Fig. 12. Power efficiency vs. number of nonzero elements: (a) Floating point matrices; (b) Binary matrices

We find that the fully associative "AP" algorithm has a computational complexity of $O(M)$ (where M is the number of nonzero elements), and its efficiency grows with the number of nonzero elements and especially with the number of nonzero elements per row. The "AP" algorithm multiplies in parallel a row vector of one matrix by the entire second matrix. As a result, the efficiency and performance of the "AP" algorithm grows with the data set size.

We show that associative implementation can offer performance benefits when multiplying sparse matrices with a limited number of predefined unique elements. Lastly, we show that AP SpMM implementation is more power-efficient than conventional GPU based solutions. This is even more evident in the case of binary matrices, thanks to the bit-oriented nature of associative processing.

Associative implementation of SpMM may benefit from further optimization, such as parallelization of matrix row processing.

ACKNOWLEDGMENT

This research was partially funded by the Intel Collaborative Research Institute for Computational Intelligence and by Hasso-Plattner-Institut.

REFERENCES

[1] A. Pinar, M. Heath. "Improving performance of sparse matrix-vector multiplication." In Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM), p. 30. ACM, 1999.

[2] C. Auth et al. "A 22nm high performance and low-power CMOS technology featuring fully-depleted tri-gate transistors, self-aligned contacts and high density MIM capacitors." VLSI Technology (VLSIT), 2012 Symposium on. IEEE, 2012.

[3] C. Foster, "Content Addressable Parallel Processors", Van Nostrand Reinhold Company, NY, 1976

[4] C. Stormon, "The Coherent Processor: an associative processor architecture and applications." In IEEE Compcon, Digest of Papers, pp. 270-275., 1991.

[5] D. Bowler, T. Miyazaki, M. Gillan. "Parallel sparse matrix multiplication for linear scaling electronic structure calculations." Computer physics communications 137, no. 2 (2001): 255-273.

[6] D. Hentrich et al., "Performance evaluation of SRAM cells in 22nm predictive CMOS technology," IEEE International Conference on Electro/Information Technology, 2009.

[7] E. Im, K. Yelick. Optimizing the performance of sparse matrix-vector multiplication. University of California, Berkeley, 2000.

[8] F. Pollack, "New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address)", MICRO 32, 1999

[9] G. Blelloch, "Vector Models for Data-Parallel Computing", MIT Press, 1990.

[10] G. Goumas, et al. "Performance evaluation of the sparse matrix-vector multiplication on modern architectures", The Journal of Supercomputing 50.1 (2009): 36-77.

[11] G. Qing, X. Guo, R. Patel, E. Ipek, E. Friedman. "AP-DIMM: Associative Computing with STT-MRAM," ISCA 2013.

[12] H. Li et al. "An AND-type match line scheme for high-performance energy-efficient content addressable memories," IEEE Journal of Solid-State Circuits , vol. 41, no. 5, pp. 1108 – 1119, May 2006.

[13] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-280/specifications

[14] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-285/specifications

[15] I. Scherson et al., "Bit-Parallel Arithmetic in a Massively-Parallel Associative Processor", IEEE Transactions on Computers, Vol. 41, No. 10, October 1992

[16] 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.

[17] J. Bolz, I. Farmer, E. Grinspun, and Peter Schröoder. "Sparse matrix solvers on the GPU: conjugate gradients and multigrid." In ACM Transactions on Graphics, vol. 22, no. 3, pp. 917-924. ACM, 2003.

[18] J. Davis, E. Chung. "SpMV: A memory-bound application on the GPU stuck between a rock and a hard place" Microsoft Technical Report, 2012.

[19] J. Kurzak, D. Bader, J. Dongarra, "Scientific Computing with Multicore and Accelerators", CRC Press, Inc., 2010.

[20] J. Sun, G. Peterson, O. Storaasli. "Sparse matrix-vector multiplication design on FPGAs." In Field-Programmable Custom Computing Machines, 15th Annual IEEE Symposium on FCCM, pp. 349-352, 2007.

[21] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: a tutorial and survey," IEEE Journal of Solid-State Circuits, vol. 41, no. 3, pp. 712 – 727, March 2006

[22] L. Yavits, "Architecture and design of Associative Processor for image processing and computer vision", MSc Thesis, Technion – Israel Institute of technology, 1994, available at http://webee.technion.ac.il/publication-link/index/id/633

[23] L. Yavits, A. Morad, R. Ginosar, "Computer Architecture with Associative Processor Replacing Last Level Cache and SIMD Accelerator", IEEE Transactions on Computers, 2013

[24] L. Yavits, A. Morad, R. Ginosar, "The effect of communication and synchronization on Amdahl's law in multicore systems", Parallel Computing Journal, 2013

[25] L. Zhuo, V. Prasanna. "Sparse matrix-vector multiplication on FPGAs." In Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays, pp. 63-74. ACM, 2005.

[26] M. Baskaran, R. Bordawekar. "Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies." IBM Research Report, RC24704 (W0812-047) (2008).

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

[28] N. Bell, M. Garland. "Implementing sparse matrix-vector multiplication on throughput-oriented processors." In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, p. 18. ACM, 2009.

[29] N. Bell, M. Garland. "Efficient sparse matrix-vector multiplication on CUDA", Vol. 20. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[30] O. Beaumont, et al. "A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers)", IEEE Transactions on Computers, 50.10 (2001): 1052-1070.

[31] O. Beaumont, et al. "Matrix multiplication on heterogeneous platforms", IEEE Transactions on Parallel and Distributed Systems, 12.10 (2001): 1033-1051.

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

[33] Q. Zhu, et al. "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware", IEEE HPEC 2013

[34] R. Boisvert et al., "The Matrix Market: A web resource for test matrix collections", Quality of Numerical Software, Assessment and Enhancement, pp. 125–137 (http://math.nist.gov/MatrixMarket)

[35] R. Kieckhager, C. Pottle, "A processor array for factorization of unstructured sparse networks", IEEE Conf. on Circuits and Computers, 1982, pp. 380-383.

[36] R. Vuduc, "Automatic performance tuning of sparse matrix kernels." PhD diss., University of California, 2003.

[37] S. Borkar. "Thousand Core Chips: A Technology Perspective," Proc. ACM/IEEE 44th Design Automation Conf. (DAC), 2007, pp. 746-749.

[38] S. Sengupta, M. Harris, Y. Zhang, J Owens. "Scan primitives for GPU computing." In Graphics Hardware, vol. 2007, pp. 97-106. 2007.

[39] S. Toledo, "Improving the memory-system performance of sparse-matrix vector multiplication." IBM Journal of research and development 41, no. 6 (1997): 711-725.

[40] S. Williams et al., "Optimization of sparse matrix–vector multiplication on emerging multicore platforms." Parallel Computing 35, no. 3 (2009): 178-194.

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

[42] X. Liu, M. Smelyanskiy, "Efficient sparse matrix-vector multiplication on x86-based many-core processors", International conference on supercomputing, ACM, 2013.

[43] Y. Fung, "Associative Processor Architecture - a Survey", ACM Computing Surveys Journal (CSUR), Volume 9, Issue 1, March 1977, Pages 3 – 27

[44] Y. Saad, A. Malevsky. "PSPARSLIB: A portable library of distributed memory sparse iterative solvers", Tech. Rep. UMSI 95/180, University of Minnesota, 1995.