Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware

Qiuling Zhu, Tobias Graf, H. Ekin Sumbul, Larry Pileggi, Franz Franchetti
Dept. of Electrical and Comp. Eng., Carnegie Mellon University, Pittsburgh, PA, USA
Email: {qiulingz,hsumbul,pileggi,franzf}@ece.cmu.edu

***Abstract*—This paper introduces a 3D-stacked logic-in-memory (LiM) system to accelerate the processing of sparse matrix data that is held in a 3D DRAM system. We build a customized content addressable memory (CAM) hardware structure to exploit the inherent sparse data patterns and model the LiM based hardware accelerator layers that are stacked in between DRAM dies for the efficient sparse matrix operations. Through silicon vias (TSVs) are used to provide the required high inter-layer bandwidth. Furthermore, we adapt the algorithm and data structure to fully leverage the underlying hardware capabilities, and develop the necessary design framework to facilitate the design space evaluation and LiM hardware synthesis. Our simulation demonstrates more than two orders of magnitude of performance and energy efficiency improvements compared with the traditional multi-threaded software implementation on modern processors.**

***Index Terms*—3D-Stacked DRAM; TSV; Logic-in-Memory; CAM; Sparse Matrix Matrix Multiplication**

I. Introduction

Graphs are fundamental data structures used in many data analysis problems (e.g. WWW graph, social networks) [14]. These problems are important enough to support architecture investments to surpass the traditional computing which has reached the limit for increasing performance without an increase in power [25], [17]. It is widely accepted to exploit the duality between sparse matrix and graph to solve graph algorithms. For example, generalized sparse matrix-sparse matrix multiplication (SpGEMM) is a key primitive for graph algorithms such as breadth-first search and shortest-path algorithms [4]. However, the development of sparse matrix algorithms poses numerous challenges due to their sparse and irregular data structures and the low ratio of flops to memory access. When running them on modern computers, most of time and energy are spent on moving data rather than on computation. Further, today's systems are power limited, exacerbating the problem.

An effective approach to address these challenges requires to co-optimize the algorithm, architecture and hardware. As shown in Fig. 1, we propose a logic-in-memory (LiM) accelerated 3D DRAM system that is customized for accelerating the notoriously hard sparse data problems. The proposed computing system has logic layers stacked in between DRAM dies which communicate with each other vertically using through silicon vias (TSVs) [19], [26]. Facilitated by the sub-20nm regular pattern construct based circuit design [22], we dedicate the logic layer by tightly integrating the application-specific logic with the embedded memory blocks, resulting in

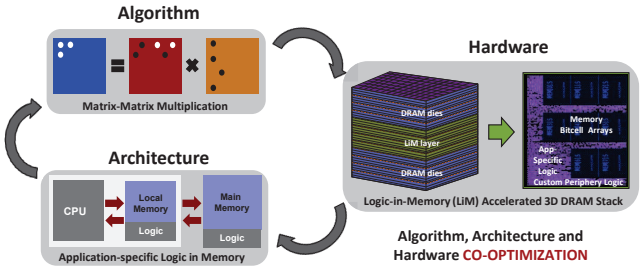

Fig. 1. 3D DRAM-LiM Stacked System Optimized for Graph Algorithms.

an 3D-stacked logic-in-memory accelerator layer (i.e., LiM-layer). The 3D-stacked architecture and its TSV capabilities, along with the localized application-specific embedded memory specialization on the LiM layer, can optimize the system to a level that is impossible with general purpose computing or configurable hardware computing.

In implementation we fully exploit the rich silicon estate on the LiM layer and the fast and dense TSVs to hold as many as active DRAM pages to increase the DRAM cache capacity [19], [26]. To design the LiM layer in an energy-efficient manner and minimize the thermal hotspots in the 3D stack [20], we carefully analyze the inherent data storage and access patterns that are existing in the algorithms, and build a logic-enhanced content-addressable memory (CAM) architecture by taking advantage of its internal parallel matching capabilities, to accelerate the index alignment of SpGEMM algorithm [23]. Moreover, we revise the sparse data structure and algorithm to adequately leverage the customized hardware.

Design co-optimization from algorithm to hardware gives rise to a huge design tradeoff space. The 3D DRAM modeling framework is based on the CACTI-3DD for fast design space exploration [9]. We have also developed an end-to-end LiM design framework for the automated hardware synthesis of the customized LiM layer with user control of architectural parameters [29]. Our experimental results demonstrate orders of magnitude of performance and energy improvements compared with the multi-threaded software SpGEMM implementations on modern CPU.

II. Sparse Data Format and SpGEMM Algorithm

Efficient implementation of sparse matrix operations requires careful choice of data structures to avoid the storage and operations on zeros. In our approach, different choices of sparse data structures and algorithms not only determine the data distribution in the 3D DRAM, but also the smart

978-1-4799-1365-7/13/$31.00 ©2013 IEEE

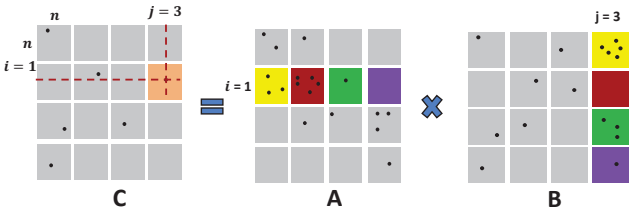

Fig. 2. 2D Block Decomposition Based SpGEMM

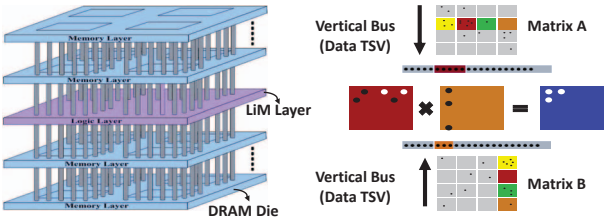

Fig. 3. 3D DRAM-LiM Stacked Architecture for SpGEMM Implementation.

LiM layer design and the inter-layer data communication. In this section, we will exploit the suitable SpGEMM algorithms and the sparse data formats that match to the customized architecture and hardware characteristics.

We use a 2-dimensional (2D) block data decomposition of the matrices based on the SRUMMA algorithm [15], which is modified from the Cannon's algorithm for distributed memory architectures [6]. As shown in Fig. 2, the matrix A, matrix B, and resulting matrix C are tiled into smaller blocks. We assume each block is $n \times n$-size and has nnz number of non-zero (nnz) elements. The resulting block $C(i, j)$ is computed from the i^{th} block row of the matrix A and j^{th} block column of the matrix B. This sequential block access order preserves good data locality and minimizes the DRAM row miss. It also allows an easy-scheduled 3D DRAM access to fully utilize the TSV bandwidth. However, such matrix decomposition will cause overhead to store and access the block addresses (e.g., block meta-data). In Section IV we will exploit efficient meta-data storage formats to minimize the overhead cost.

For the multiplication of any two blocks, we use the column-by-column SpGEMM algorithm introduced in [5]. As shown in Fig. 4, the basic idea of the algorithm is to compute a column i of block C as a linear combination of the block A that are specified by the nonzero elements in the same column i of the block B, and it constructs one column C at a time. This algorithm avoids the unnecessary operations of data which is not at the intersections. The algorithm requires the random access to columns of the matrix block A, for which we can use the compressed sparse column (CSC) data storage format that allows column-wise random access [3]. However, the CSC format requires space complexity of $O(n + nnz + nnz)$ and the dependency on the matrix block dimension n causes too much overhead when the matrices get very sparse [5]. Therefore we adopt the doubly compressed sparse column (DCSC), which was used for hypersparse matrix data storage when $nnz < n$ [5]. DCSC reduces the space complexity from $O(n)$ to $O(nnz)$ but still allows column-wise random access. Besides, we use Coordinate (COO) storage format with the space complexity of $O(nnz)$ for the second source matrix

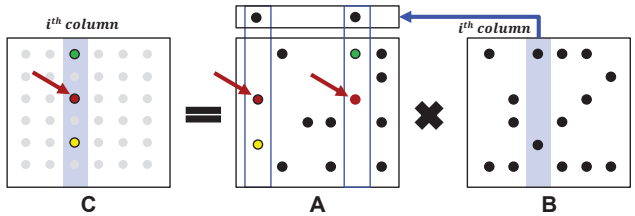

Fig. 4. Intra-Block Column-by-Column SpGEMM Algorithm

block B which allows sequential access in the algorithm [3].

III. 3D-Stacked LiM based SpGEMM Design

In this section we will introduce the mapping the SpGEMM algorithm to the 3D stacked LiM hardware. As illustrated in Fig. 3, DRAM dies and an enhanced LiM layer are stacked vertically. To compute each resulting matrix block, we transfer the entire two source matrix blocks to the LiM layer through the TVS link. The dense, short, and fast TSVs are able to transfer the two source matrix blocks to the LiM layer in a few clock cycles. To match the high throughput requires building the efficient dedicated hardware on the LiM layer that can perform equivalently high performance SpGEMM operations. Next we will first introduce the naive hardware mapping of the SpGEMM algorithm and then the details of the smart CAM based LiM design.

A. Naive Hardware Mapping of SpGEMM on LiM Layer

In most software implementations of the column-by-column algorithm, a heap (priority queue) of size $nnz(B(:, i))$ is used to construct the column $C(:, i)$ using the multiway merge [5]. It stores one nonzero element for each intersected columnA, and repeatedly extracts the minimum row index element for computation and then inserts the next non-zero element from the same column [5]. To map the algorithm to the hardware on the LiM layer, one can implement the heap array using the SRAM FIFO [12]. However, in a typical dual-port SRAM design, the heap insert (or delete) operations required to extract-min incur uncertain delay, resulting in a complicated datapath pipeline design. Alternatively, one can replace the SRAM arrays with the shift registers which allows one-cycle heap operation but at the expense of the hardware cost [21]. In our approach we try to avoid the use of heap to eliminate both of its latency and cost overhead by slightly revising the algorithm and directly assembling the resulting column C using a content addressable memory (CAM) hardware structure. As we will describe in detail next, the CAM structure is similar to the sparse accumulator (SPA) used in the current Matlab algorithm [24]. However, while SPA uses the space and time complexity of $O(n)$, our CAM structure only uses space $O(nnz)$ and time complexity of $O(flops)$, thus being more hardware-friendly in terms of overall cost and performance.

B. Smart CAM-Based SpGEMM LiM Design

The approach stems from the observation that the numerical values of the non-zero elements are always accompanied with their row and/or column index information, and the SpGEMM operations involve intensive index matching operations. Such

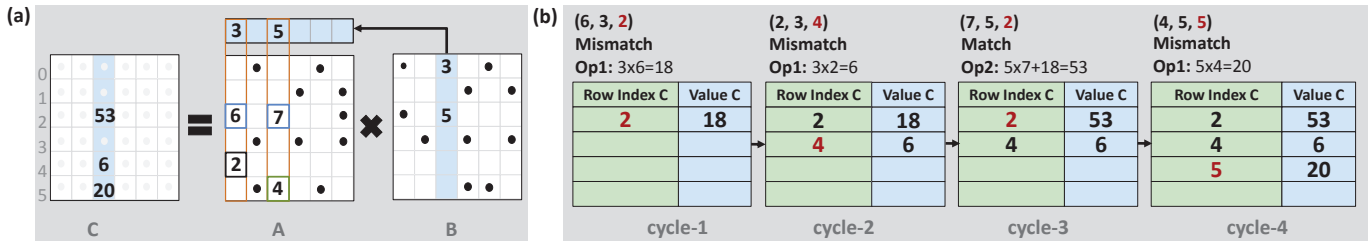

Fig. 5. CAM Operation Example.

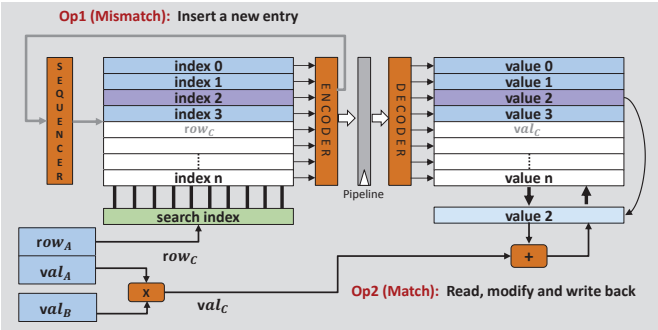

Fig. 6. SpGEMM-CAM Functional Architecture.

unique algorithm characteristics match well to the CAM architecture, in which each word has an embedded comparator and data is associated with a key, rather than an address.

Customized CAM design. A CAM is a memory that implements a parallel comparison or matching function using dedicated comparison circuitry [23], [18]. It offers a single cycle throughput by simultaneously comparing the input search data against a table of stored data, and returns the address (or the active wordlines) of the matched data. We customize the CAM structure for assembling the resulting matrix C in the column-by-column SpGEMM algorithm. The CAM array is used to construct one resulting column C at a time and it is composed of three parts: an index CAM array to store the row index of each resulting non-zero element; a data SRAM array to store its numerical value; and a set of control and arithmetic logic sitting around the CAM array. There is no need to store the column indices as they are always the same at a time.

In implementation we revise the column-by-column algorithm by processing the intersected columns of block A one column after another sequentially until the computation of a resulting column C is finished. Fig. 6 shows the proposed CAM architecture and the fundamental CAM operations. Assuming a new arithmetic operation is finished which multiplies a non-zero element (row_A, val_A) with its intersected val_B, it results in an intermediate non-zero element C, $val_C = val_A \times val_B$, with row index row_C the same as row_A. Next, CAM first compares row_C with all the existing index array entries. If it successfully finds an existing index (e.g., $index2$) that matches with row_C, it indicates that val_C should contribute to the same numerical data $value2$. In this situation, the matched CAM matchline will automatically activate the corresponding wordline of the data array, reading out $value2$, accumulating with val_C, and writing back to the same location. However, in a different scenario where the CAM fails to match the input row_C with any of the existing index array entries, a mismatch signal will be triggered, which will then activate the CAM decoder (a sequencer) to move its pointer forward by one step, and store row_C and val_C as a new CAM entry. Fig. 5 shows the CAM operations to compute the third column of the example matrix C, which eventually involves three mismatch operations and one match operation.

Horizontal CAM and vertical CAM. The proposed CAM block is used to assemble one resulting matrix column at a time. In a parallel implementation it requires implementing multiple CAM blocks on the LiM layer, one for each column and each CAM block is labeled with the corresponding column index for identification purpose. Therefore, for each multiplication result, we first need to select the right CAM block based on its column index. To achieve this, we implement another orthogonal (vertical) CAM which stores the column indices of the all the existing (horizontal) CAM blocks and compare them with the incoming column index simultaneously, identifying the right one in one cycle. Eventually the resulting CAM system comprises multiple horizontal CAM blocks and one vertical CAM block, further improving the throughput.

Overall SpGEMM LiM architecture Fig. 7 presents the overall diagram of a single SpGEMM core which multiplies the source matrix block A and B, and assembles the resulting matrix block C. As we can see, besides the vertical and horizontal CAM blocks that we described above for assembling the resulting matrix C, there are also another two LiM blocks, that is, LiM-1 and LiM-2 for the storage and accessing of the source matrix A and B, respectively. As matrix B is stored in the COO format, its accessing logic as in LiM-2 is simply a sequential counter. Overall the LiM-2 outputs the row index (row_B), which serves as the intersected column index to search in matrix block A (col_A); as well as the column index (col_B), which is the same as col_C and to be stored into the vertical CAM. Matrix block A as in LiM-1 is in the DCSC format, and it involves five data arrays which are called AUX, JC, CP, IR and NUM, respectively [5], requiring smarter logic-in-memory accessing strategies. For example, the AUX and CP arrays always access two consecutive entries simultaneously, the difference between which indicates the range of entries to check in their next data arrays (i.e., JC and IR). Therefore we implement them using our previous proposed parallel access memory, which allows to read out two adjacent SRAM entries in one clock cycle without too much overhead [28]. The JC array stores the non-zero column indices, and the access of which requires to search (access

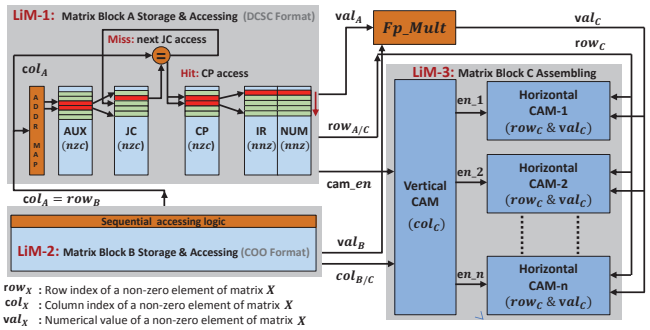

Fig. 7. Functional Diagram for a Single SpGEMM Core

and compare) the data entries in a pre-determined range sequentially until the required column index is found. To achieve this, it has a comparator integrated with the memory array. If a hit is returned from the comparator, it indicates that the input column index is found, and then the CP entry at the same location will be automatically activated, which points to the location of the intersected non-zero elements of that column. Otherwise, it continues to search in the JC array until the required column index is found or the end of the array is reached. Eventually LiM-1 outputs the row indices of the intersected elements A (row_A) if they exist, serving also as row_C, and to be stored into the horizontal CAMs. LiM-1 and LiM-2 also produce the numerical values val_A and val_B, which are multiplied by a floating-point multiplication unit (*fp-mult*) and the result (val_C) is stored into the SRAM part of the horizontal CAMs. Moreover, LiM-1 also produces an enabling signal (cam_en) when the intersected elements are successfully found, and activates the CAM-based LiM-3 which is disabled otherwise. Similarly, each horizontal CAM is also controlled by an enable signal and only one of them is activated each time to minimize the power consumption.

The proposed LiM-1 and LiM-2 blocks appear like ordinary SRAM blocks and provide a "dense" view of the sparse source matrices, which produce the intersected non-zero elements to the arithmetic unit continuously; but essentially they have dedicated logic operations integrated in the memory abstraction, aiming to identify the intersected non-zero elements and perform necessary computation before returning the data. The CAM-based LiM-3 block is also tightly integrated with the enhanced logic functionalities, including a read-modify-write unit for CAM match operations as well as a sequential access decoder for CAM mismatch operations. Eventually it can assemble one multiplication result at a cycle, significantly improving the throughput compared with the original heap based implementations, as we will evaluate in Section IV

LiM design automation framework. The actual LiM implementation is a large design tradeoff problem that involves both application and architectural parameters. Moreover, the fine grained memory-logic-mixing design further increases the design complexity. LiM is originated from sub-20nm pattern construct based design [22], from which we have developed a LiM synthesis tool that can efficiently synthesize the LiM blocks including both of the memory and the logic functionalities in one shot [29]. This method facilitates the fast design space exploration and automated hardware generation.

C. 3D-Stacked DRAM System Modeling

To fully take advantage of the TSV-based 3D DRAM, we exploit the fine-grained rank-level 3D die-stacked DRAM architecture [9], which re-partitions the DRAM arrays by allowing individual memory banks to be stacked in a 3D fashion. And we used CACTI-3DD, a 3D die-stacked DRAM modeling tool to explore 3D DRAM design space in terms of area, power and bandwidth [9]. The design details can be found in another accompanying work [27]. Here we exploit one of the optimal 3D-DRAM design points, that is, a 8 Gb 3D-stacked DRAM system composed of four stacked DRAM dies [2], [13]. Each DRAM die implements 16 banks of DDR3 DRAM and each has a separate 2 KB active row buffer. The banks within a 3D vertical rank share the 512-bit data TSV bus [9]. The selected 3D-DRAM design offers 350 GB/s memory bandwidth at 50% of area efficiency while consuming 12 Watts of power, serving as a suitable 3D-DRAM system to accelerate the proposed CAM-based SpGEMM design that balances the performance and cost.

IV. Analysis and Results

In this section, we will evaluate the proposed CAM-based SpGEMM implementation based on the 3D DRAM-LiM stacked architecture, and compare with the conventional designs. The benchmark matrices are from the University of Florida sparse matrix collection [10].

CAM design evaluation. The high throughput of a CAM comes at the cost due to its internal parallel comparisons. Therefore, it is important to keep the CAM size as small as possible to save area and power. Fortunately, the sparsity of the matrix allows to use very small-size CAM blocks. Fig. 8 plots the required maximum entries of a horizontal CAM from the SpGEMM experiments of 15 different benchmark matrices. We see that most matrices only require CAM with less than 64 entries and their sizes further decrease when the block dimension gets smaller. This is reasonable as theoretically the horizontal CAM size is determined by the number of non-zero elements per column of the resulting matrix block. To better understand the statistical distribution of the CAM sizes, we simulate near one hundred of benchmark matrices. Fig. 9 (a) and (b) plot the distribution histograms for the maximum horizontal and vertical CAM sizes required to calculate the non-zero matrix blocks. Note the y-axis in logarithm scale is the statistical count of the matrix blocks. We see the CAM size statistics follow an exponentially declining trend. This experiment demonstrates that the CAM-based method imposes little demands for the embedded-memory storage on the LiM layer, making this approach feasible.

In Fig. 10 we compare the CAM design performance with the naive heap-based implementation with both CSC and DCSC formalizations. We measure the *cycles_per_flop*, which is defined as the average clock cycles per arithmetic operation and less *cycles_per_flop* indicates higher utilization of the computational resources and higher throughput. We see that

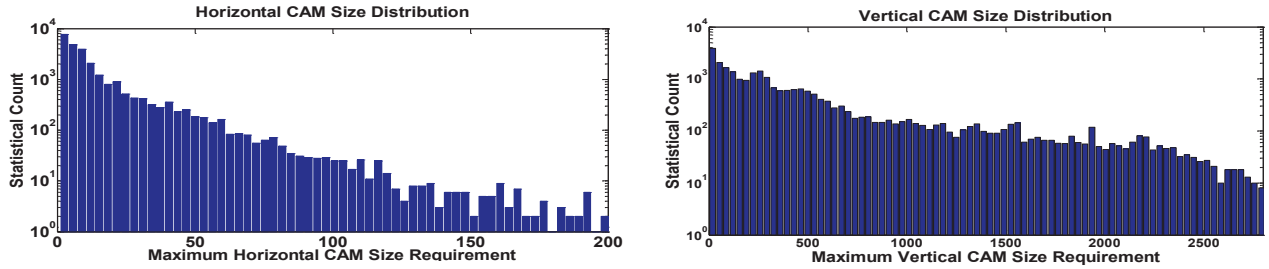

Fig. 9. CAM Size Distribution Histogram.

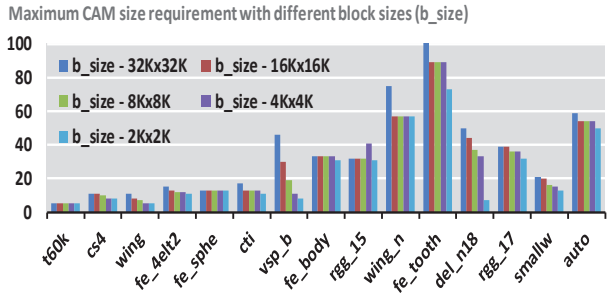

Fig. 8. CAM Size Evaluation.

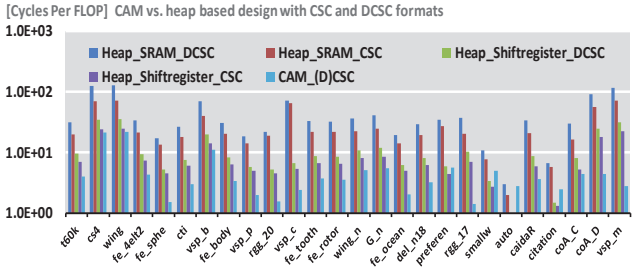

Fig. 10. CAM Performance Evaluation.

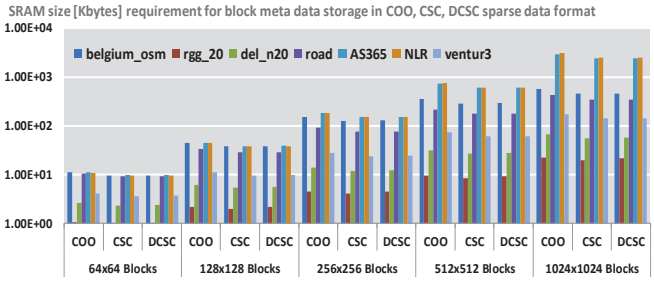

Fig. 11. Meta Data Storage Evaluation.

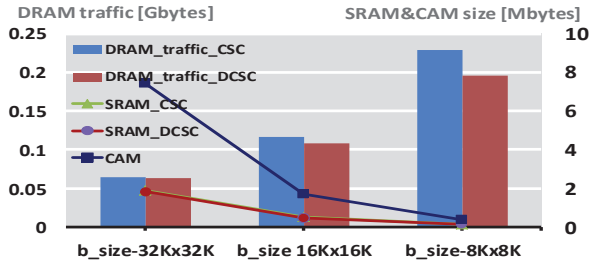

Fig. 12. Analysis of Block Size Choices.

the CAM design consumes about one order of magnitude less *cycles_per_flop* compared with SRAM-based heap design due to its one-cycle matching capability. While shift register-based heap design is much more efficient than SRAM-based design, it still performs worse than CAM. It is also noticed that the use of DCSC format has an adverse effect on the performance as it requires an extra search in the JC array due to its doubly compressed nature [5]. However, in our CAM design, the JC array search latency can be overlapped with the CAM operation as we process the intersected columnA sequentially, which allows us to proceed the DCSC access to the next intersected columnA simultaneously when we process the current one. This co-optimization of the algorithm and hardware delivers extremely high computing throughput.

Data format and block dimension evaluation. Next we analyze the sparse data storage format and block dimension choices. Fig. 12 shows that SpGEMM with smaller-size blocks has increasing data traffic between the LiM layer and the DRAM dies. If we decompose a matrix into $N_B \times N_B$ blocks, it is easy to derive that the total data traffic is proportional to N_B. Therefore we see a scaling of two of the data traffic when we scale the block size (b-size) from $32K \times 32K$, $16K \times 16K$, to $8K \times 8K$. On the other hand, the smaller block dimension saves SRAM and CAM storage on the LiM layer as a trade-off. One way to alleviate the burden of both is to exploit a more efficient data storage format. As seen in Fig. 12, DCSC data format turns out to be effective especially for a finer 2D decomposition where the blocks get smaller and sparser.

An overhead cost associated with the 2D block decomposition is the storage and the access of the block meta data that contains the DRAM address information of each non-zero block. Due to the sparsity of the source matrices, many blocks after decomposition are empty and require no operations. Therefore, the meta data itself can be stored in an extra sparse matrix of size $N_B \times N_B$. Fig. 11 plots the meta data storage requirement under different data formats. We see that the CSC and DCSC formats are equally superior to the COO format, which implies that the meta data matrix is not as sparse as the data matrix and it is not necessary to use the DCSC format to avoid its adverse effect on the accessing latency.

Performance and energy evaluation. Next we evaluate the performance and energy efficiency of the 3D LiM-based SpGEMM. We run Intel Math Kernel Library (MKL) Sparse Basic Linear Algebra Subprograms (BLAS) Routines on Intel Xeon machines for comparison [1]. We use the Sniper multi-core simulator integrated with McPAT and the modified USIMM DRAM simulator for system power evaluation [7], [16], [8], [11]. Fig. 13 presents the comparison of FLOPS (FLoating-point Operations Per Second) of the two systems where we vary the block dimension of the 3D LiM implementations and the thread counts in Intel MKL implementations.

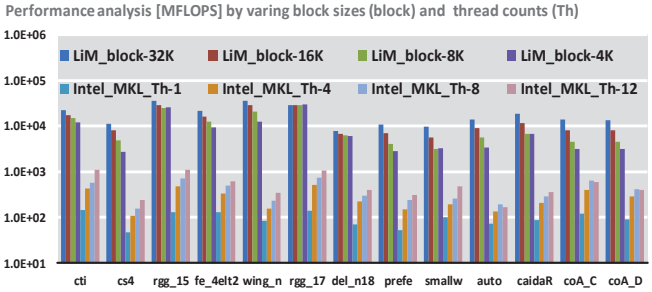

Fig. 13. SpGEMM Performance Analysis.

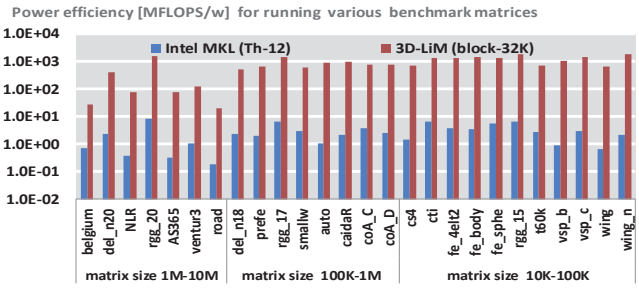

Fig. 14. SpGEMM Power Efficiency Evaluation.

We see that the 3D-LiM performance decreases with the decreasing block dimensions due to the scaling factor of N_B in TSV traffic, and the performance of the software implementation increases with the increasing thread counts. However, the 3D-LiM performs better in all simulated sceneries. Fig. 14 demonstrates that the proposed 3D LiM design achieves up to three orders of magnitude power efficiency improvement for a wide variety of benchmark matrix simulations.

V. Conclusion

This paper presents a 3D DRAM-LiM stacked hardware primitive that targets at accelerating SpGEMM-based sparse graph problems. The novelty lies in a 3D-stacked DRAM which offers high bandwidth and low latency data transfer via TSV and a stacked LiM layer that is customized to the particular problem through a fine-grain integration of logic, CAM and SRAM. In addition, we revise the algorithms to match the underlying hardware and adapt the necessary modeling and design framework tools. The result is a transparent, power efficient hardware-accelerated device for notoriously memory-bound problems. This paper demonstrates that recent cutting-edge IC design advances create opportunities to build an extremely energy, power and performance-efficient computing platform to accelerate data intensive computing.

ACKNOWLEDGEMENT

This work was supported in part by the Intelligence Advanced Research Program Agency and Space and Naval Warfare Systems Center Pacific under Contract No. N66001-12-C-2008, Program Manager Dennis Polla. The work was also sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement No. HR0011-13-2-0007. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government. No official endorsement should be inferred.

References

[1] Intel. math kernel library. [online] http://developer.intel.com/software/products/mkl/.
[2] Samsung electronics datasheet: 4gb b-die ddr3 sdram. 2011.
[3] M. Bell, N.and Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proc. Supercomputing*, 2009.
[4] A. Buluc and J. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. *ICPP*, pages 503 – 510, 2008.
[5] A. Buluc and J. Gilbert. On the representation and multiplication of hypersparse matrices. *IPDPS*, pages 1 – 11, 2008.
[6] L. Cannon. A cellular computer to implement the kalman filter algorithm. *PhD Thesis, Montana State University*, 1969.
[7] T. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. *SC*, pages 1 – 12, 2011.
[8] N. Chatterjee. Usimm: the utah simulated memory module. *University of Utah and Intel Corp*, 2012.
[9] K. Chen, S. Li, , and et.al. Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory. *DATE*, pages 33 – 38, 2012.
[10] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1 – 1:25, 2011. [online] http://www.cise.ufl.edu/research/sparse/matrices.
[11] C. Fischer and C. McCurdy. Using pin as a memory reference generator for multiprocessor simulation. *SIGARCH Computer Architecture News*, pages 39–44, 2005.
[12] A. Ioannou and M. Katevenis. Pipelined heap (priority queue) management for advanced scheduling in high speed networks. *ICC*, pages 2043–2047, 2001.
[13] U. Kang, H. Chung, and et.al. 8 gb 3-d ddr3 dram using through-silicon-via technology. *JSSC*, pages 111 – 119, 2009.
[14] J. Kepner and J. Gilbert. *Graph algorithms in the language of linear algebra*. Society for Industrial and Applied Mathematics, 2011.
[15] M. Krishnan and J. Nieplocha. Srumma: a matrix multiplication algorithm suitable for clusters and scalable shared memory systems. *IPDPS*, 2004.
[16] S. Li, J. H. Ahn, R. Strong, J. Brockman, and et. al. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. *MICRO*, pages 469 – 480, 2009.
[17] C. Lin, Z. Zhang, and et.al. Design space exploration for sparse matrix-matrix multiplication on fpgas. *FPT*, pages 369–372, 2010.
[18] M. Lin, J. Luo, and Y. Ma. A low-power monolithically stacked 3d-tcam. *ISCAS*, pages 3318–3321, 2008.
[19] G. Loh. 3d-stacked memory architectures for multi-core processors. *ISCA*, pages 453 – 464, 2008.
[20] G. Loi, B. Agrawal, and et.al. A thermally-aware performance analysis of vertically integrated (3-d) processor-memory hierarchy. *DAC*, pages 991 – 996, 2006.
[21] S. Moon, J. Rexford, and K. Shin. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Transactions on computers*, pages 1215–1227, 2000.
[22] D. Morris, K. Vaidyanathan, and et.al. Design of embedded memory and logic based on pattern constructs,. *Symp. VLSI Technology*, 2011.
[23] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (cam) circuits and architectures: A tutorial and survey. *JSSC*, 41(3):712–727, 2006.
[24] V. Shah and J. Gilbert. Sparse matrices in matlab *p: Design and implementation. *HiPC*, pages 144 – 155, 2004.
[25] J. Stevenson, A. Firoozshahian, and et.al. Sparse matrix-vector multiply on the hicamp architecture. *ICS*, pages 195–204, 2012.
[26] D. Woo, N. Seong, and H. Lee. Heterogeneous die stacking of sram row cache and 3d dram: An empirical design evaluation. *MWSCAS*, pages 1 – 4, 2011.
[27] Q. Zhu, B. Akin, H. E. Sumbul, J. C. Hoe, L. Pileggi, and F. Franchetti. A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing. *3DIC*, 2013.
[28] Q. Zhu, L. Pileggi, and F. Franchetti. Local interpolation-based polar format sar: Algorithm, hardware implementation and design automation. *JSPS*, 71(3):297 – 312, 2013.
[29] Q. Zhu, K. Vaidyanathan, L. Pileggi, and F. Franchetti. Design automation framework for application-specific logic-in-memory blocks. *ASAP*, pages 125 – 132, 2012.