# Algorithm and Hardware Co-Optimized Solution for Large SpMV Problems

Fazle Sadi, Larry Pileggi and Franz Franchetti
Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, USA
Email: {fsadi,pileggi,franzf}@andrew.cmu.edu

***Abstract*—Sparse Matrix-Vector multiplication (SpMV) is a fundamental kernel for many scientific and engineering applications. However, SpMV performance and efficiency are poor on commercial of-the-shelf (COTS) architectures, specially when the data size exceeds on-chip memory or last level cache (LLC). In this work we present an algorithm co-optimized hardware accelerator for large SpMV problems. We start with exploring the basic difference in data transfer characteristics for various SpMV algorithms. We propose an algorithm that requires the least amount of data transfer while ensuring main memory streaming for all accesses. However, the proposed algorithm requires an efficient multi-way merge, which is difficult to achieve with COTS architectures. Hence, we propose a hardware accelerator model that includes an Application Specific Integrated Circuit (ASIC) for the muti-way merge operation. The proposed accelerator incorporates state of the art 3D stacked High Bandwidth Memory (HBM) in order to demonstrate the proposed algorithm's capability coupled with the latest technologies. Simulation results using standard benchmarks show improvements of over 100x against COTS architectures with commercial libraries for both energy efficiency and performance.**

## I. Introduction

Sparse Matrix-Vector multiplication (SpMV) can be denoted as $y = Ax + y$, where $A$, $x$ and $y$ are the sparse matrix, dense source vector and dense resultant vector respectively. It often arises as a key kernel in many big data applications. Unfortunately, this very kernel generally becomes the bottleneck for these applications as it renders very low fraction of the peak processor performance ($<$10%) [1], [2] and poor energy efficiency on commercial of-the-shelf (COTS) architectures. For large SpMV problems, where $x$ or $y$ is much bigger than the on-chip storage achievable by current technologies, this situation becomes worse.

The reason for poor performance on COTS architectures (CPU, GPU, etc.) is twofold. First is the notorious memory-wall problem [3], [4]; i.e., main memory bandwidth is already scarce in relation to available compute power. At the same time, we are trying to extract performance and efficiency for SpMV that requires high number of memory accesses for a little amount of computation. This is a technological barrier. The second, and probably more important reason is that COTS architectures are built upon conventional memory hierarchy that expects spatial and temporal locality in the data. However, due to sparsity, large SpMV problems are almost devoid of both spatial and temporal locality. This renders conventional memory hierarchy in COTS platforms inherently unsuitable for SpMV. It is an architectural problem that we constrain ourselves with when we select a COTS platform for SpMV implementation.

Nevertheless, researchers looked into the architectural constraints and tried to fit SpMV on COTS architectures by numerous techniques. As an example, for CPU, researchers have tried mitigating the impact of the sparsity in matrix structure by employing sophisticated storage formats, such as DIA, ELL, BCM-CSR [5], or by using intensive preprocessing methods such as register blocking, matrix reordering [6], [7]. For GPU, researchers have explored many sophisticated approaches, such as fine grained parallel decomposition [1], model based auto-tuning [8] and transformation of matrix representation and tiling to increase temporal locality [6]. However, performance and efficiency gain through these methods are still far from what is achieved for dense matrix operations.

**Contributions:** In this work, we propose a solution for large SpMV problems that achieves 100x improvement over conventional architectures both in performance and efficiency. The key steps for this solution are as follows.

*1)* To solve the architectural problem, we approach SpMV from opposite direction. That means, instead of constraining ourselves with the architectural resources, we first select the algorithm that has the most suitable data transfer characteristics for large SpMV and, afterwards, build the required implementation platform. We propose an algorithm, namely Two-Step, that ensures DRAM streaming for all data access (i.e. full utilization of memory bandwidth) and gets rid of the excessive data transfer generally required for SpMV implementations on COTS architectures. However, for this proposed algorithm to achieve high performance and efficiency, we require multi-way merge operation on a large number of long sorted lists. As this is difficult to efficiently achieve with CPU or GPU, we present an Application Specific Integrated Circuit (ASIC) as the computation platform for the proposed Two-Step algorithm.

*2)* To address the memory-wall problem, we consider state of the art 3D stacked main memory, namely High Bandwidth Memory (HBM) [9], and interposer [10] technology. The extreme bandwidth provided by HBM relaxes the memory-wall issue significantly. Hence, we present a SpMV accelerator model that integrates the ASIC for Two-Step algorithm to HBM through interposer. In this work, we only constrain ourselves by the available technology, rather than commercial feasibility, and demonstrate what is achievable by the presented algorithm/hardware co-optimized solution.

The remainder of the paper is organized as follows. Sec. II and Sec. III demonstrate the proposed algorithm and the hardware accelerator accordingly. In Sec. IV we evaluate various SpMV algorithms and provide the rationale behind our proposed solution. Experimental results are presented in

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

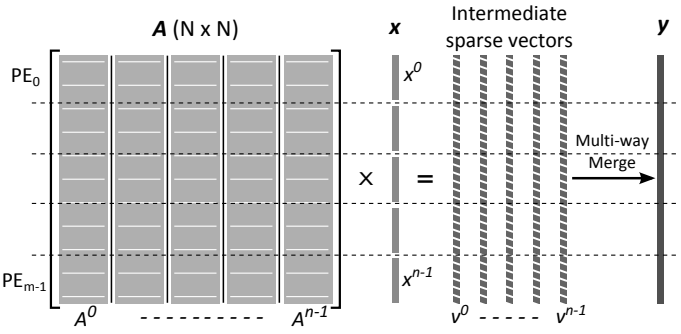


Fig. 1: Two-Step SpMV algorithm.

Sec. V. Lastly, Sec. VI concludes this work.

## II. Proposed Algorithm

In this work we propose an algorithm for large SpMV, namely Two-Step, that fundamentally depends on column-wise matrix blocking and multi-way merge operation. As the name suggests, it works in two distinct steps that are depicted in Figure 1. A pseudocode of Two-Step SpMV is shown in Pseudocode 1. For this algorithm, initially we divide the source vector ($x$) into several segments and the source matrix ($A$) into several vertical stripes of the same width. The stripes of $A$ are stored in a row major format and we select compressed sparse row (CSR) [7] for our implementation. The width of each segment of $x$ directly corresponds to the size of fast on-chip storage. From algorithm point of view solely, we only need 1D partitioning (column blocks) of the matrix. However, 2D partitioning, as in [11], [12], is required for parallelization and proper bandwidth utilization in multi-core shared memory scenario. Hence, we resort to 2D partitioning by further dividing the matrix stripes horizontally into blocks.

**Pseudocode 1:** Two-Step algorithm for large SpMV.

```
STEP 1
for k = 0 to n − 1 do
    Stream in Matrix Column Block A^k
    u ← 0
    for All rows A^k_{i,:} with nnz > 0 do
        for Each non-zero A^k_{i,j} in A^k_{i,:} do
            Random access to vector segment x^k
            u_i ← A^k_{i,j} * x^k_j + u_i
        end
    end
    Sparsify u to v^k
    Stream out v^k
end
STEP 2
for i = 0 to N − 1 do
    for k = 0 to n − 1 do
        Stream in v^k
        y_i ← y_i + v^k_i [Multiway Merge]
    end
end
Stream out y
```

**Step 1.** In the first step of the algorithm, one source vector segment is streamed to the on-chip storage from main memory. Then, the corresponding matrix stripe is streamed from the main memory. Afterwards, matrix elements are multiplied with the corresponding vector elements and accumulated to any existing partial results within the same matrix stripe. Each Processing Element (PE) takes care of this process for its allocated portion of the matrix stripe and all PEs share the source vector segment residing in shared on-chip storage. When this process completes, we get an intermediate vector, $v^k$, for $k^{th}$ matrix stripe and each PE streams back its corresponding portion of $v^k$ to main memory. This intermediate vector is sparse and, therefore, positional index is stored along with the value. As we traverse the matrix elements in row-wise direction within the stripe, elements of $v^k$ are sequentially generated in ascending order of the row pointers of the matrix elements. Therefore, each sparse intermediate vector is sorted according to its elements' indices (keys), which is vitally important for the second step of this algorithm. In the first step, we continue this process for all the matrix stripes and end up with $n$ sparse intermediate vectors residing in DRAM.

**Step 2.** In the second step, all intermediate vectors ($v^k$s) are streamed back from main memory to PEs, where essentially a multi-way merge operation on these sorted lists ($v^k$s) is conducted to construct the resultant vector $y$. Each PE only merges a portion of the intermediate vectors as shown in Figure 1.

The main contribution of Two-Step algorithm is that it guarantees complete DRAM streaming access while getting rid of the excess source vector data transfer due to matrix partitioning. Each element of the source vector makes only a single trip from the DRAM to on-chip storage, which is generally not true for large SpMV implementations on conventional architectures. We will discuss this in detail in Sec. IV.

So far, we remained oblivious to the computation requirement for the SpMV algorithms. Our proposed Two-Step algorithm can theoretically be implemented on CPU or GPU. Unfortunately, for large SpMV problems Two-Step requires a challenging task of merging thousands of sorted lists, where each list has millions of elements. This large multi-way merge is compute-bound [13], [14] and difficult to accomplish with COTS architectures efficiently due to bad scaling behavior. However, with custom hardware design, it is possible to achieve efficient and scalable multi-way merge for large number of long lists. Therefore, as a solution, in this work we develop a co-optimized ASIC hardware accelerator as the implementation platform for our proposed Two-Step algorithm.

## III. Proposed Hardware Accelerator

As an implementation platform for Two-Step algorithm, we develop a hardware accelerator in this work, which is depicted in Figure 2. For main memory we use multiple HBMs [15]. This state of the art 3D stacked memories can provide extreme bandwidth (on the order of TB/s with multiple stacks). The entire system sits on an interposer base to provide wide high speed channel between main memory and computation cores. HBM along with interposer significantly relaxes the *memory wall* issue for memory bound problems like SpMV. However, it should be noted that HBM is not mandatory for Two-Step algorithm to be useful. This algorithm ensures the proper use

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

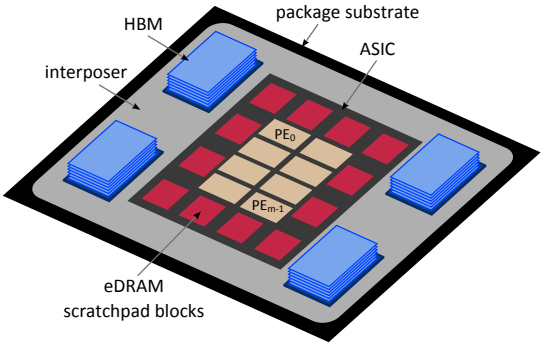


Fig. 2: Proposed SpMV accelerator model configuration.

of main memory bandwidth regardless of what technology is used. We have modeled the accelerator with HBM mainly to demonstrate what can be achieved by using this state of the art technology without constraining ourselves to the monetary cost of the accelerator.

For the on-chip storage, the fast random access storage needed for the algorithm, we use Embedded DRAM (eDRAM) scratchpad. It has higher density and lower leakage compared to Static Random Access Memory (SRAM). Furthermore, eDRAM uses many more banks and small page size that allow low-power operation at modest area penalty [16] and can provide high random access bandwidth [17].

The PEs consist of two different types of cores. The first type, namely Multiply-Accumulate (MA) core, takes care of the step 1 of the algorithm. It comprises multiple sets of pipelined floating point multipliers and adders connected in series as depicted in Figure 3. Computation in step 1 related to a matrix stripe can be taken care by the MA cores of multiple PEs. The number of total MA cores depend on the random access bandwidth of the on-chip eDRAM and streaming bandwidth of the DRAM. As eDRAM has many small banks and the probability of bank conflict for accessing $x^k$ (due to sparsity in $A^k$) is low, the on-chip storage can be shared among the MA cores without introducing many stalls in the pipeline.

The other type of core, namely Multi-way Merge (MM) core, handles the second step. A combination of radix-sort network and binary tree based pipelined merge-sort network constitute the MM core. Full hardware details of the MM core is beyond the scope of this paper. The MM cores also utilize a small portion of the eDRAM scratchpad to store DRAM page size level data blocks for each list ($v^k$). This is done to fully amortize the DRAM page opening cost each time a load request for list data is issued. The custom MM ASIC can provide high enough throughput to saturate the streaming bandwidth of HBM. Moreover, several MM cores of multiple PEs can work in parallel and independently merge different portions of the intermediate vectors.

## IV. Evaluation of Large SpMV Algorithms

SpMV has a high memory access to compute ratio and, hence, data transfer characteristics is one of the most important aspects of SpMV algorithms. In this section, to evaluate our proposed algorithm, we explore the basic differences in data transfer characteristics for different families of SpMV algorithms while remaining oblivious to the compute requirements. We assume the Disk Access Machine (DAM) model [18] with

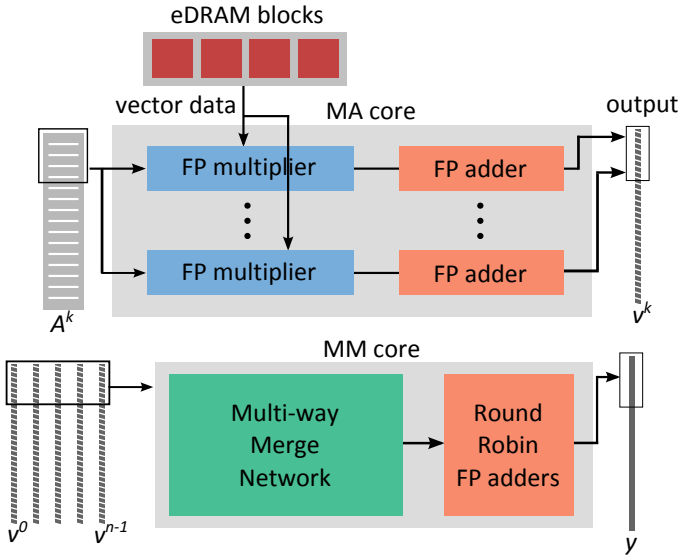


Fig. 3: Configuration of a Processing Element (PE).

two levels of memory hierarchy, on-chip storage (fast access) and main memory (slow access with block transfer). We present a high level mathematical model for data transfer between the fast and slow memory levels in a single core scenario, however, our analysis is applicable to multi-core shared memory scenarios too.

Since an efficient SpMV kernel should be memory-bound [1], generally the measure of success of an SpMV algorithm is the fraction of peak bandwidth that can be achieved. In this study, we mainly consider the algorithms that achieves full main memory (DRAM) streaming access (i.e. utilizes peak bandwidth). Therefore, our measure of success would be the execution time that is inversely proportional to the volume of data transferred between the memory hierarchy levels.

Large sparse matrices are generally partitioned to avoid random access to DRAM and for parallelization. Figure 4 shows such a $N \times N$ matrix which is partitioned into $m \times n$ blocks. We consider a square matrix to simplify the calculation without any loss of generality. Matrix $A$ has a total of at most $hN$ non-zeros, where $h$ is the average number of non-zeros per row. We also define $S_m$ as the average size of matrix element (including meta-data) and $S_v$ as the size of source and resultant vector elements.

Independent of the matrix block data storage format and computation method, there are basically two ways to traverse the matrix blocks. As shown in Figure 4, one way is to traverse the blocks in row-major direction and another is in column-major direction. We name the family of algorithms following row-major block traversal as $Algo^r$ and other that follows column-major block traversal as $Algo^c$. Our proposed Two-Step algorithm falls under the family of $Algo^c$.

For $Algo^r$, as used in [19], the matrix block and relevant segment of the source vector ($x$) can be streamed to the chip from DRAM and discarded after multiply-accumulate operations for the block. Thus, the entire source vector has to be streamed from DRAM for traversal of one row of the matrix blocks and cannot be stored on chip for re-use due to large size. On the other hand, the resultant vector portion needs to be stored on-chip for random access since it is updated while

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

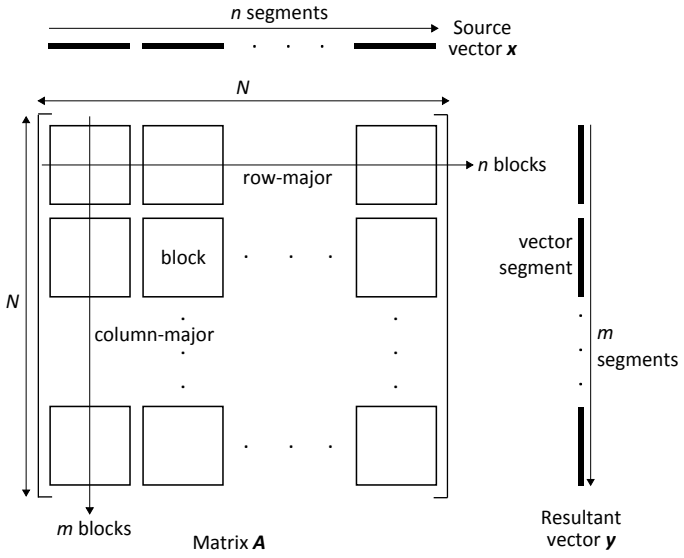


Fig. 4: Matrix blocking and different ways of block traversal.

a row of matrix blocks is traversed. Therefore, resultant vector segment ($N/m$) is dictated by the on-chip storage size. Table I shows the data transfer amount between on-chip memory and DRAM for $Algo^r$. For example, to compute on each block row of the matrix we need to stream $hNS_m/m$ amount of matrix data from DRAM to on-chip storage. For the entire computation on matrix $A$, $hNS_m$ amount of matrix data is streamed into the chip from DRAM. From Table I, we can express the total data transfer for $Algo^r$ ($D^r$) as given in Equation 1.

TABLE I: Data transfer volume between DRAM and on-chip memory for $Algo^r$ (row-major block access)

| Data source (direction w.r.t. chip) | Data per block row | Total data |
|---|---|---|
| Matrix (in) | $hNS_m/m$ | $hNS_m$ |
| Source vector (in) | $NS_v$ | $mNS_v$ |
| Resultant vector (out) | $NS_v/m$ | $NS_v$ |

$$D^r = hNS_m + NS_v + mNS_v \tag{1}$$

For $Algo^c$, similar to $Algo^r$, matrix blocks can be streamed from DRAM to chip. However, in this case we need to randomly access the segment of source vector $x$ as we traverse the $m$ matrix blocks in column-major direction. Therefore, the on-chip storage is occupied by the source vector segment. Another important difference from $Algo^r$ is that each column of matrix block will produce an intermediate resultant vector. In the worst case (when there is no reduction), an intermediate vector will have the same number of elements as the total Number of Non-Zeros (NNZs) in all the matrix blocks in that column. We need to update the intermediate resultant vector as we move to the next column of matrix blocks. As the intermediate resultant vector becomes more dense each time updated, storing it on chip is not feasible. Now, there are few ways we can update. One way is to keep the intermediate vector in DRAM and update its elements by randomly accessing DRAM, but this will cause inefficient bandwidth use and make SpMV latency bound. Another way is to stream the entire intermediate result to chip every time we traverse a new column of matrix blocks. However, even though guaranteeing DRAM streaming, large amounts of redundant data will have to be transferred between the chip and DRAM in this method. From a data transfer perspective solely, a better way is to stream out the $n$ intermediate vectors (one for each column block of $A$) to DRAM as they are produced. Afterwards, we have to stream all intermediate vectors out from DRAM to chip and apply reduction operations on them to get the final resultant vector $y$. Table II presents the data transfers for the operations in $Algo^c$. Further, we can deduce the total data transfer for $Algo^c$ ($D^c$) as shown in Equation 2.

TABLE II: Data transfer volume between DRAM and on-chip memory for $Algo^c$ (column-major block access).

| Data source (direction w.r.t. chip) | Data per block column | Total data |
|---|---|---|
| Matrix (in) | $hNS_m/n$ | $hNS_m$ |
| Source vector (in) | $NS_v/n$ | $NS_v$ |
| Intermediate vec. (out & in) (assuming no reduction) | $2hNS_v/n$ | $2hNS_v$ |
| Resultant vector (out) | - | $NS_v$ |

$$D^c = hNS_m + NS_v + (2h+1)NS_v \tag{2}$$

From Equation 1 and Equation 2 we see that the first two terms ($hNS_m$ and $NS_v$) are identical. The third terms in the equations make these algorithms distinct from each other. For $Algo^r$, $mNS_v$ appears due to reading the entire source vector from DRAM $m$ times. On the other hand, $(2h+1)NS_v$ represents the intermediate results of $Algo^c$ that make a round-trip from chip to DRAM. By investigating further, we can see that the key factors are $m$ and $(2h+1)$ for $Algo^r$ and $Algo^c$ respectively. If $m < (2h+1)$, then $Algo^r$ is preferable as it will have less data transfer than $Algo^c$. Otherwise, $Algo^c$ is preferable when $m > (2h+1)$.

Next, we derive $m$ from system configuration, matrix dimension ($N$) and sparsity ($h$). For any system with DRAM capacity of $C^{DRAM}$, the largest matrix dimension $N$ that it is able to handle, given $h$, can be calculated from Equation 3. We assume that the matrix, source vector and resultant vector occupy the main memory entirely.

$$C^{DRAM} = \overset{\text{matrix}}{hNS_m} + \overset{\text{source vector}}{NS_v} + \overset{\text{resultant vector}}{NS_v}$$
$$\Rightarrow N = \frac{C^{DRAM}}{(hS_m + 2S_v)} \tag{3}$$

Now, $Algo^r$ needs to store $\frac{1}{m}$th portion of the resultant vector in the on-chip storage. Therefore, we can express the capacity of the on-chip memory $C^{chip}$ as

$$C^{chip} = \frac{NS_v}{m}. \tag{4}$$

Replacing $N$ from Equation 3 to Equation 4 gives us closed form expression for $m$ as shown in Equation 5.

$$m = \frac{C^{DRAM}}{C^{chip}} \frac{S_v}{(hS_m + 2S_v)} \tag{5}$$

From the above equation we see that $m$ is directly proportional to the ratio of slow(DRAM) and fast(on-chip) memory size. This ratio for typical systems is generally much greater than the sparsity of matrices for SpMV problems, rendering

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

TABLE III: Speedup with $Algo^c$ over $Algo^r$ on typical systems.

| System | On-chip/DRAM (MB/GB) | Peak read bandwidth | N | $m$ | Execution time (s) $Algo^r$ | Execution time (s) $Algo^c$ | Gain with $Algo^c$ |
|---|---|---|---|---|---|---|---|
| GPU (Tesla GP100 w/ HBM2) | 4/16 | 366 GB/s | 250M | 500 | 2.76 | 0.05 | 50x |
| FPGA (Stratix 10 w/ HBM2) | 16/32 | 512 GB/s | 500M | 250 | 2.00 | 0.08 | 26x |
| Desktop (Skylake Core i7) | 8/64 | 34 GB/s | 1B | 1e3 | 236.7 | 2.35 | 100x |
| Server (Haswell Xeon E7) | 45/1500 | 102 GB/s | 23B | 4.2e3 | 7.7e3 | 12.6 | 417x |

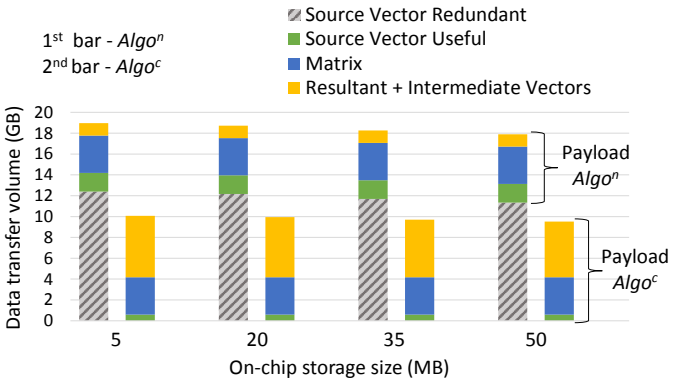


Fig. 5: Data transfer comparison between $Algo^n$ and $Algo^c$.

$m$ to be much larger than $2h+1$. For example, $h$ can be in the order of $1 \rightarrow 10$ for large sparse matrices (e.g. social network graphs such as *Twitter lists*, *YouTube* from KONECT library [20]). On the other hand, $m$ is in the order of thousands for systems capable of handling large SpMV problems.

For a number of practical systems, we explore the value of $m$, largest matrix dimension $N$ and execution time of both the algorithms (assuming data is transferred at peak bandwidth) for $h = 3$. Table III summarizes our findings. We see that $m$ is in the order of hundreds for systems will relatively low DRAM size. For systems with large DRAM capacity, $m$ is in the order of thousands which is lot larger than $(2h+1) = 7$. Therefore, $Algo^r$ needs much more data transfer than $Algo^c$, specially for large SpMV problems. For example, a Haswell server with 1.5 TB DRAM can operate on 23B$\times$ 23B matrix with average NNZ per row of 3. If $Algo^r$ is used, the source vector of 23B elements needs to be transferred from DRAM to chip $m = 4200$ times (600x larger than $2h+1$), whereas for $Algo^c$ the source vector is transferred only once. As the overhead, due to intermediate results, for $Algo^c$ is lot less than transferring source vector $m$ times, we get 417x improvement in execution time with $Algo^c$. Furthermore, $m$ is directly proportional to matrix dimension $N$, which makes $Algo^r$ less scalable than $Algo^c$. Therefore, our analysis shows that column-major matrix block traversal based SpMV algorithms, such as our proposed Two-Step algorithm, are preferable for large SpMV problems.

Another category of algorithm, $Algo^n$, follows a naive approach of computing resultant vector elements $y_i = \sum_{j=0}^{N-1} A_{ij}x_j$ entirely for increasing $j$ and randomly accesses the entire source vector in DRAM to load $x_j$. Here, $i$ and $j$ are the row and column indices of matrix $A$. Unlike $Algo^r$, we don't need to stream the entire source vector $m$ times and one could argue that $Algo^n$ has the minimum amount of data volume that actually visits the computation core. However, this algorithm is generally latency bound as it requires random access in DRAM. More importantly, for $Algo^n$, cache-line level data is transferred to chip for almost each vector element, which has little re-use (even for large on-chip storage) due to sparsity in matrix $A$. Thus, huge amount of redundant data is transferred at low random access DRAM bandwidth. An example data transfer volume between DRAM and on-chip for $Algo^n$ vs $Algo^c$ is shown in Figure 5 for a matrix of size 80M$\times$80M, $h = 3$ and 64B cache line. The on-chip storage size is varied to demonstrate its insignificant effect on the overall data transfer. It can be seen that the redundant data (striped gray region) in $Algo^n$ makes the overall data transfer significantly greater than $Algo^c$. The solid colored regions represent the payload, namely the data that actually take part in computation.

One interesting observation in Figure 5 is that the payload for $Algo^c$, hence for Two-Step, is consistently greater than the latency bound naive approach ($Algo^n$). It is because the intermediate vectors ($v^k$s) have to make a full round trip to DRAM for $Algo^c$. Nonetheless, for iterative algorithms, e.g. PageRank [19], it is possible to eliminate the increase in payload due to the round trip of $v^k$s by overlapping the two steps in proposed algorithm. Due to lack of space we will skip that discussion in this work.

## V. Experimental Results

To compare our modeled accelerator against standard benchmarks, we first ran a comprehensive design space exploration under various constraints. A set of system specifications and constrains is given in Table IV and the relevant design space parameters are given in Table V. All the design parameters in Table V are varied to compute the energy efficiency and performance metrics of the system for a typical problem of matrix size 80M$\times$80M and $h = 3$. Afterwards, an optimum design point is picked that meets all the constraints. HBM specifications [15] and CACTI-3DD [21] are used to simulate 32nm technology node 3D stacked HBM. Destiny tool [22] is used to simulate eDRAM at 32nm node. The RTL design of the MA and MM core are synthesized using 28nm commercial standard cell library. The memory synthesis methodology in [23] is used to generate SRAM blocks for our design.

TABLE IV: System specifications and constraints for design space exploration and simulation of the SpMV accelerator.

| |
|---|
| 8GB per HBM, 8 HBMs, 512GB/s bandwidth per HBM |
| 1KB Page, Interposer area (including HBM) limit 200mm$^2$ |
| MA core power limit 100mW, MM core power limit 40mW |

We compare the efficiency and performance of our proposed accelerator with several state of the art COTS architectures.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

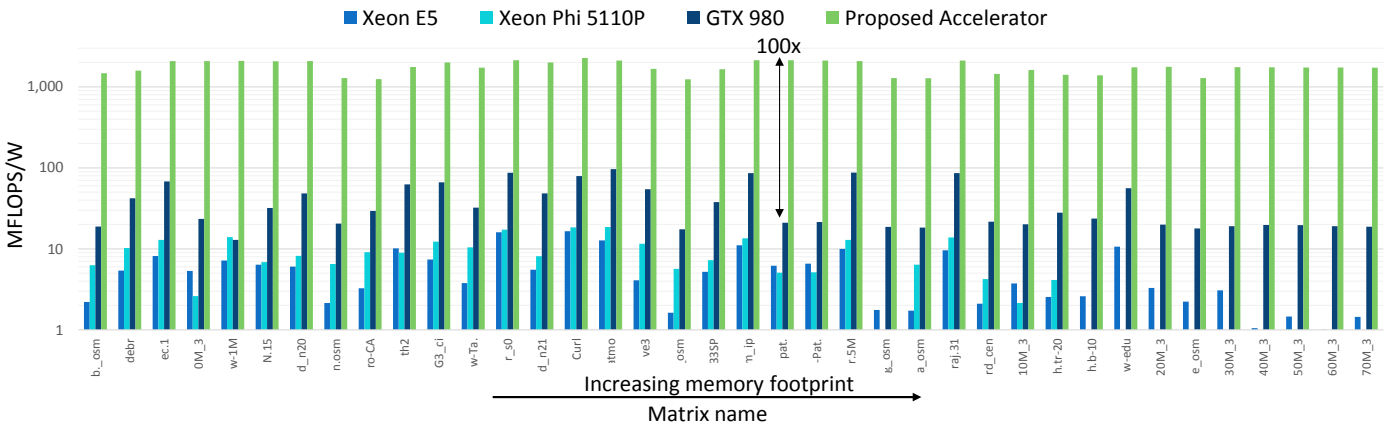


Fig. 6: Energy efficiency comparison results among different architectures.

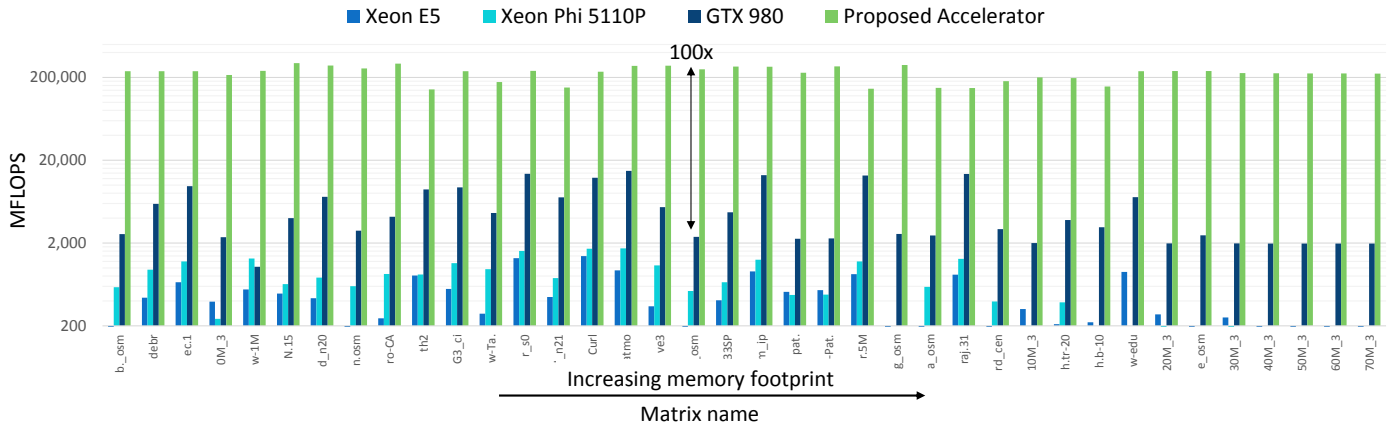


Fig. 7: Performance comparison results among different architectures.

TABLE V: Design space parameters for SpMV accelerator.

| |
|---|
| eDRAM block size 64KB→16MB |
| Number of multiplier-adder pairs in a MA core 1→128 |
| Number of lists that can be merged by single MM core 32→4096 |
| Number of MA cores 1→32, Number of MM cores 1→24 |

For test matrices, we use the sparse matrix collection from University of Floria [24]. The matrix dimensions vary from 1M to 70M with 1 to 5 NNZ per row. We also use some of our randomly generated sparse matrices to introduce worst case scenarios in terms of matrix patterns. As a benchmark we implemented SpMV using Intel Math Kernel Library (MKL) on dual socket Xeon E5-2620 (22nm) CPU and Xeon Phi 5110P (22nm) co-processor. Both of the architectures have 30MB last level cache (LLC). The peak bandwidth is 102GB/s for the CPU and 352GB/s for the co-processor. Furthermore, Nvidia CUDA Sparse (cuSPARSE) library is used to implement SpMV on GTX-980 GPU with a peak bandwidth of 224GB/s. The energy efficiency and performance are measured for the benchmarks and simulated for the proposed accelerator. Results are shown in Figure 6 and Figure 7. As we can see, the CPU (Xeon E5) has the poorest metrics while the co-processor (Xeon Phi 5110P) deliver mediocre metrics on average. The GPU (GTX-980) has the best efficiency and performance among the COTS architectures in most cases. Nevertheless, the proposed accelerator consistently achieves two orders of magnitude better performance and efficiency than the GPU. This significant achievement is possible due to the combination of proper algorithm that ensures good data transfer characteristics and custom hardware with state of the art technologies that is co-optimized for the target algorithm.

We expect our simulated results to match real implementation as the simulator load assumptions and power estimations are pessimistic. Nonetheless, in reality, it is important to properly tune the interface between main memory and the ASIC to ensure that DRAM page size block data is transfered for each read and write requests. This is critical for maximally utilizing DRAM bandwidth and fully amortizing DRAM page opening costs.

## VI. Conclusion

Large SpMV problems pose a unique set of challenges that conventional memory hierarchy based COTS architectures are inherently not suitable for. To achieve significant performance and efficiency improvement, we have proposed an algorithm and hardware co-optimized solution in this work. The proposed algorithm is developed only to minimize the data traffic to/from main memory and ensure full streaming access while remaining oblivious to the compute requirements. The custom hardware is finely tuned to address the algorithm's computational requirement, which is mainly a large multi-way merge network. Experimental results confirm the substantial potential of our proposed approach in gaining significant improvement for SpMV kernel over state of the art COTS solutions.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

## ACKNOWLEDGMENT

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

[2] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, 2003, aAI3121741.

[3] Q. Zhu, B. Akin, H. Sumbul, F. Sadi, J. Hoe, L. Pileggi, and F. Franchetti, "A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing," in *3D Systems Integration Conference (3DIC), 2013 IEEE International*, Oct 2013, pp. 1–7.

[4] Q. Wu, K. Rose, J. Q. Lu, and T. Zhang, "Impacts of though-dram vias in 3d processor-dram integrated systems," in *3D System Integration, 2009. 3DIC 2009. IEEE International Conference on*, Sept 2009, pp. 1–6.

[5] J. B. White III and P. Sadayappan, "On improving the performance of sparse matrix-vector multiplication," in *High-Performance Computing, 1997. Proceedings. Fourth International Conference on*. IEEE, 1997, pp. 66–71.

[6] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining," *Proceedings of the VLDB Endowment*, vol. 4, no. 4, pp. 231–242, 2011.

[7] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix–vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009.

[8] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," *SIGPLAN Not.*, vol. 45, no. 5, pp. 115–126, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1837853.1693471

[9] J. Standard, "High bandwidth memory (hbm) dram," *JESD235*, 2013.

[10] K. Cho, H. Lee, H. Kim, S. Choi, Y. Kim, J. Lim, J. Kim, H. Kim, Y. Kim, and Y. Kim, "Design optimization of high bandwidth memory (hbm) interposer considering signal integrity," in *2015 IEEE Electrical Design of Advanced Packaging and Systems Symposium (EDAPS)*, Dec 2015, pp. 15–18.

[11] B. Vastenhouw and R. H. Bisseling, "A two-dimensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM Rev.*, vol. 47, no. 1, pp. 67–95, Jan. 2005. [Online]. Available: http://dx.doi.org/10.1137/S0036144502409019

[12] W. Yang, K. Li, Z. Mo, and K. Li, "Performance optimization using partitioned spmv on gpus and multicore cpus," *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2623–2636, Sept 2015.

[13] L. Rutkowski, M. Korytkowski, R. Scherer, R. Tadeusiewicz, L. A. Zadeh, and J. M. Zurada, *Artificial Intelligence and Soft Computing: 12th International Conference, ICAISC 2013, Zakopane, Poland, June 9-13, 2013, Proceedings, Part I ... / Lecture Notes in Artificial Intelligence)*. Springer Publishing Company, Incorporated, 2013.

[14] Y. Shan, T. Wu, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang, "Fpga and gpu implementation of large scale spmv," in *Application Specific Processors (SASP), 2010 IEEE 8th Symposium on*, June 2010, pp. 64–70.

[15] B. Black, "Die stacking is happening in mainstream computing," *Additional Conferences (Device Packaging, HiTEC, HiTEN, & CICMT)*, vol. 2014, no. DPC, pp. 001 183–001 206, 2014. [Online]. Available: http://dx.doi.org/10.4071/2014DPC-keynote_w1_black

[16] J. Poulton, "An embedded DRAM for CMOS ASICs," in *Advanced Research in VLSI, 1997. Proceedings., Seventeenth Conference on*. IEEE, 1997, pp. 288–302.

[17] M. Jacunski, D. Anand, R. Busch, J. Fifield, M. Lanahan, P. Lane, A. Paparelli, G. Pomichter, D. Pontius, M. Roberge, and S. Sliva, "A 45nm soi compiled embedded dram with random cycle times down to 1.3ns," in *Custom Integrated Circuits Conference (CICC), 2010 IEEE*, Sept 2010, pp. 1–4.

[18] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, no. 9, pp. 1116–1127, Sep. 1988. [Online]. Available: http://doi.acm.org/10.1145/48529.48535

[19] T. Haveliwala, "Efficient computation of pagerank," Stanford InfoLab, Technical Report 1999-31, 1999. [Online]. Available: http://ilpubs.stanford.edu:8090/386/

[20] J. Kunegis, "Konect: The koblenz network collection," in *Proceedings of the 22Nd International Conference on World Wide Web*, ser. WWW '13 Companion. New York, NY, USA: ACM, 2013, pp. 1343–1350. [Online]. Available: http://doi.acm.org/10.1145/2487788.2488173

[21] K. Chen, S. Li, N. Muralimanohar, J.-H. Ahn, J. Brockman, and N. Jouppi, "CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory," in *Design, Automation Test in Europe (DATE)*, 2012, pp. 33–38.

[22] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie, "Destiny: A tool for modeling emerging 3d nvm and edram caches," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2015*, March 2015, pp. 1543–1546.

[23] H. E. Sumbul, K. Vaidyanathan, Q. Zhu, F. Franchetti, and L. Pileggi, "A synthesis methodology for application-specific logic-in-memory designs," in *Proceedings of the 52Nd Annual Design Automation Conference*, ser. DAC '15. New York, NY, USA: ACM, 2015, pp. 196:1–196:6. [Online]. Available: http://doi.acm.org/10.1145/2744769.2744786

[24] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663