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 off-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 multi-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 off-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
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

---

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

1. **STEP 1**
   1. for \( k = 0 \) to \( n - 1 \) do
   2. Stream in Matrix Column Block \( A^k \)
   3. \( u ← 0 \)
   4. for All rows \( A^k_{i,j} \) with \( \text{nnz} > 0 \) do
      5. for Each non-zero \( A^k_{i,j} \) in \( A^k_{i,j} \) do
         6. Random access to vector segment \( x^k \)
         7. \( u_i ← A^k_{i,j} \times x^k_j + u_i \)
      end
   end
   8. Sparsify \( u \) to \( v^k \)
   9. Stream out \( v^k \)
   end

2. **STEP 2**
   1. for \( k = 0 \) to \( N - 1 \) do
      2. for \( k = 0 \) to \( n - 1 \) do
         3. Stream in \( v^k \)
         4. \( y_i ← y_i + v^k_i \) [Multiway Merge]
      end
   end
   11. Stream out \( y \)

**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.
We assume the Disk Access Machine (DAM) model [18] with the number of total MA cores depend on the random access 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 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 \( \text{Algo}^r \) and other that follows column-major block traversal as \( \text{Algo}^c \). Our proposed Two-Step algorithm falls under the family of \( \text{Algo}^c \).

For \( \text{Algo}^r \), 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...
Another way is to stream the entire intermediate result to chip. For each column block of \( A \), the intermediate resultant vector \( v \) becomes more dense each time updated, storing it on the next column of matrix blocks. As the intermediate resultant vector accumulates, the entire resultant vector \( y \) will have less data transfer than the sparsity of matrices for SpMV problems, rendering 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 \( \text{Algo}^2 \). Further, we can deduce the total data transfer for \( \text{Algo}^2 (D^c) \) as shown in Equation 2.

**TABLE II: Data transfer volume between DRAM and on-chip memory for \( \text{Algo}^2 \) (column-major block access).**

<table>
<thead>
<tr>
<th>Data source (direction w.r.t. chip)</th>
<th>Data per block row</th>
<th>Total data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source vector (in)</td>
<td>( hNS_m / n )</td>
<td>( hNS_m )</td>
</tr>
<tr>
<td>Intermediate vec. (out &amp; in)</td>
<td>( 2hNS_v / m )</td>
<td>( 2hNS_v )</td>
</tr>
<tr>
<td>Resultant vector (out)</td>
<td>-</td>
<td>( NS_v )</td>
</tr>
</tbody>
</table>

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

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 \( \text{Algo}^2 \), \( hNS_m \) 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 \( \text{Algo}^2 \) 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 \( \text{Algo}^2 \) and \( \text{Algo}^c \) respectively. If \( m < (2h + 1) \), then \( \text{Algo}^c \) is preferable as it will have less data transfer than \( \text{Algo}^2 \). Otherwise, \( \text{Algo}^c \) is preferable when \( m > (2h + 1) \).

For \( \text{Algo}^c \), similar to \( \text{Algo}^2 \), 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 \( \text{Algo}^2 \) 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 (NZNs) 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 \( \text{Algo}^2 \). Further, we can deduce the total data transfer for \( \text{Algo}^2 (D^c) \) as shown in Equation 2.

**TABLE I: Data transfer volume between DRAM and on-chip memory for \( \text{Algo}^c \) (row-major block access).**

\[
D^c = hNS_m + NS_v + mNS_v \tag{1}
\]
TABLE III: Speedup with Algo\textsuperscript{c} over Algo\textsuperscript{r} on typical systems.

<table>
<thead>
<tr>
<th>System</th>
<th>On-chip/DRAM (MB/GB)</th>
<th>Peak read bandwidth</th>
<th>N</th>
<th>m</th>
<th>Execution time (s)</th>
<th>Gain with Algo\textsuperscript{c}</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU (Tesla GP100 w/ HBM2)</td>
<td>4/16</td>
<td>366 GB/s</td>
<td>250M</td>
<td>500</td>
<td>2.76</td>
<td>0.05</td>
</tr>
<tr>
<td>FPGA (Stratix 10 w/ HBM2)</td>
<td>16/32</td>
<td>512 GB/s</td>
<td>500M</td>
<td>250</td>
<td>2.00</td>
<td>0.08</td>
</tr>
<tr>
<td>Desktop (Skylake Core i7)</td>
<td>8/64</td>
<td>34 GB/s</td>
<td>1B</td>
<td>1e3</td>
<td>236.7</td>
<td>2.35</td>
</tr>
<tr>
<td>Server (Haswell Xeon E7)</td>
<td>45/1500</td>
<td>102 GB/s</td>
<td>23B</td>
<td>4.2e3</td>
<td>7.7e3</td>
<td>12.6</td>
</tr>
</tbody>
</table>

![Fig. 5: Data transfer comparison between Algo\textsuperscript{r} and Algo\textsuperscript{c}.](image)

For a number of practical systems, we explore the value of \( m \), largest matrix dimension \( N \) and execution time of both of 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 capable of handling large SpMV problems.

For \( m \) 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.

Another category of algorithm, Algo\textsuperscript{r}, follows a naïve 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\textsuperscript{c}, we don’t need to stream the entire source vector \( m \) times and one could argue that Algo\textsuperscript{r} 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\textsuperscript{r}, 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\textsuperscript{r} vs Algo\textsuperscript{c} is shown in Figure 5 for a matrix of size 80M×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\textsuperscript{r} makes the overall data transfer significantly greater than Algo\textsuperscript{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\textsuperscript{c}, hence for Two-Step, is consistently greater than the latency bound naïve approach (Algo\textsuperscript{r}). It is because the intermediate vectors \( (v^k) \) have to make a full round trip to DRAM for Algo\textsuperscript{c}. Nonetheless, for iterative algorithms, e.g. PageRank [19], it is possible to eliminate the increase in payload due to the round trip of \( v^k \) 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×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 200mm² 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.
For test matrices, we use the sparse matrix collection from University of Florida [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.
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


