A 7.3 M Output Non-Zeros/J, 11.7 M Output Non-Zeros/GB Reconfigurable Sparse Matrix-Matrix Multiplication Accelerator

Dong-hyeon Park†, Subhankar Pal*, Siying Feng†, Paul Gao†, Jielun Tan*, Austin Rovinski†, Shaolin Xie†, Chun Zhao†, Aporna Amarnath†, Timothy Wesley†, Jonathan Beaumont†, Kuan-Yu Chen†, Chaitali Chakrabarti†, Michael Taylor†, Trevor Mudge†, David Blaauw†, Hun-Seok Kim*, Ronald Dreslinski†

*University of Michigan, Ann Arbor, MI †University of Washington, Seattle, WA ‡Arizona State University, Tempe, AZ

Abstract—A Sparse Matrix-Matrix multiplication (SpMM) accelerator with 48 heterogeneous cores and a reconfigurable memory hierarchy is fabricated in 40 nm CMOS. The compute fabric consist of dedicated floating-point multiplication units, and general-purpose Arm Cortex-M0 and Cortex-M4 cores. The on-chip memory reconfigures as scratchpad or cache depending on the phase of the algorithm. The memory and compute units are interconnected with synthesizable coalescing crossbars for efficient memory access. The 2.0 mm×2.6 mm chip exhibits 12.6×(8.4×) energy efficiency gain, 11.7×(77.6×) off-chip bandwidth efficiency gain and 17.1×(36.9×) compute density gains against a high-end CPU (GPU) across a diverse set of synthetic and real-world power-law graph based sparse matrices.

Index Terms—Sparse matrix multiplier, synthesizable crossbar, decoupled access-execution, reconfigurability and accelerator

I. INTRODUCTION

The emergence of big data and massive social networks have led to increased importance of graph analytics and machine learning workloads. One of the fundamental kernels that drives these workloads is matrix multiplication. Traditionally, matrix multiplication workloads focused on performing linear algebra operations on dense matrices that depended primarily on high compute throughput. However, modern datacenter workloads are large and extremely sparse, where a majority of their contents are zeros. Thus, there has been an increase in attention towards matrix multiplication algorithms that target large sparse matrices, such as the adjacency matrix of Facebook friendships which is of size 1.08 B×1.08 B but with only 0.0003% Non-Zero Elements (NZEs) [1]. Sparse matrix-matrix multiplication is a significant building block of multiple algorithms prevalent in graph analytics, such as breadth-first search [2] [3], graph contraction [4], peer pressure clustering [5], Markov clustering [6], and triangle counting [7]. Other computing applications, such as color intersection searching [8], context-free grammar parsing [9], and finite element simulations based on domain decomposition [10], also rely heavily on sparse matrix-matrix multiplication.

Since the number of zero elements in a sparse matrix largely outnumber the Number of Non-Zeros (NNZs), it is prudent to store them in compressed formats. The Compressed Sparse Row (CSR) format is a standard for storing sparse matrices in graph analytics, scientific computation, etc [11]. It represents an $N \times N$ sparse matrix using three arrays – values, column-indices and row-pointers, with a total storage overhead of 2·

Fig. 1: High-level overview of sparse matrix-matrix multiplication using inner product and outer product methods.

$\text{NNZ} + N + 1$ elements. Compressed Sparse Column (CSC) is the transposed form of CSR, where the latter two arrays are replaced by row-indices and column-pointers, respectively.

Standard inner-product based multiplication algorithms are not suitable for processing extremely large, sparse matrices, because the majority of computations are wasted on processing zero-value elements. Efficient sparse matrix-matrix multiplication (SpMM) that focuses only on the NZEs greatly improve the performance of such workloads. The SpMM kernel is quintessentially memory-bound rather than compute-bound, due to poor data locality and compute-to-communication ratio. Thus, accelerating SpMM requires eliminating redundant memory accesses and maximizing data reuse.

The most common implementation of matrix-matrix multiplication is the inner product method, as shown in Figure 1. In the inner product method, a row of the first operand is multiplied by the column of the second operand to produce a

\[ C = A \times B \]

Where $A$ and $B$ are matrices and $C$ is the resulting matrix.

The inner product method is simple and straightforward, but it suffers from poor cache locality and limited parallelism.

The outer product method, on the other hand, is more complex but provides better cache locality and higher parallelism.

\[ C = A \otimes B \]

Where $A$ and $B$ are matrices and $C$ is the resulting matrix.

The outer product method is more complex but provides better cache locality and higher parallelism.
single element in the result matrix. While this approach works efficiently for dense matrices, once the matrices become too sparse, a significant portion of the runtime is spent on index matching the two operands to find nonzero elements with the same row or column indices. This results in low NNZs per byte fetched from off-chip, leading to unproductive loads. Limited on-chip storage further forces repetitive fetching of the same data, worsening the memory bottleneck.

To eliminate the wasted index matching and ensure all memory loads are productive, we employ an outer product algorithm that we first proposed in [1]. Unlike the inner product approach, the outer product approach multiplies the columns of the first operand with the rows of the second operand to generate partial product matrices that are summed together to produce the final result.

The rising importance of memory-bound problems and SpMM in particular has induced multiple works in recent years. However, many prior works only focused on improving algorithms on multi-threaded processors [12]–[14] and GPUs [15]–[17]. Some works also explored efficient SpMM implementations on FPGAs [18], [19]. A comparative study of energy-efficient SpMM implementations on contemporary platforms is done by Giefers et al. [20]. Prior fabricated designs, on the other hand, have only demonstrated sparse matrix-vector multiplication [21] and relatively high-density (≥3%) matrix-matrix multiplication with small dimensions (≤256) [22].

This paper presents the first custom SpMM accelerator that addresses the off-chip memory access bottleneck for real-world sized matrices, evaluating densities ≥0.002% and dimensions ≤120k. We evaluate our solution using output NNZ per Joule (NNZ/J) for energy efficiency, which is equivalent to throughput per Watt. For bandwidth efficiency, we calculate output NNZ per GB of fetched data (NNZ/GB), which measures throughput (NNZ/s) per bandwidth (GB/s). Our solution achieves an energy efficiency of 7.3 M output NNZ/J and bandwidth efficiency of 11.7 M NNZ/GB, achieving 11.7× and 77.6× improvements over the state-of-the-art CPU and GPU libraries, respectively.

These improvements are achieved through the following:

- A custom architecture designed to take advantage of the unique data access pattern of the outer product approach, where every memory fetch generates useful results.
- A novel synthesizable coalescing crossbar that magnifies on-chip bandwidth.
- Reconfigurable memory that can switch between cache and scratchpad to suit the different characteristics of the phases within the outer product algorithm.

The rest of the paper is organized as follows: Section II presents the outer product algorithm and how the architecture is designed to meet the needs of different kernels. Section III details the design of the coalescing crossbar and the reconfigurable memory. Section IV presents the empirical data from the chip measurements, and comparison against existing solutions. Finally, Section V summarizes our analysis and conclusions.

II. ALGORITHM AND ARCHITECTURE

Our main kernel of interest is the outer product-based sparse matrix-matrix multiplication [1], that performs \( A \times B = C \), as shown in Figure 2. The first operand, Matrix \( A \) is organized in Compressed Sparse Column (CSC) format, while the second operand, Matrix \( B \) is in Compressed Sparse Row (CSR) format. The outer product approach helps minimize redundant memory loads, and our architecture is designed to match the unique compute and memory access patterns in the different phases of the algorithm.

A. Architectural Overview

We present a high-level description of our architecture in Figure 3. The implementation of the architecture is detailed in Section III. In the multiply phase, each Processing Element (PE) multiplies an element of a column of the first operand \( A \) with a row of the second operand \( B \). The row elements of \( B \) are reused across all the PEs and thus are stored in an on-chip shared cache (Figure 3-a). For the merge phase, we switch to a pair of Arm cores, Cortex-M4 and Cortex-M0, connected by private on-chip memory. The two cores act as a single unit to stream in the results of multiply, perform merge, and store the final results to DRAM. Our studies reveal that a scratchpad
Fig. 3: High level overview of architectures suited for the multiply and merge phases. Shared cache is suited for the predictable patterns in the multiply phase, whereas a private scratchpad is better for merge.

leads to better performance than a cache for this phase, due to the irregular nature of data accesses (Figure 11).

B. Outer Product Algorithm

In this section, we describe in detail the outer product based SpMM algorithm, and how it is mapped on to the hardware.

In the multiply phase, each Processing Element (PE) multiplies a non-zero element of column $i$ of $A$ with all non-zero elements of row $i$ of $B$ to produce one Partial Product Matrix (PPM) row. Each NZE is fetched only once. The PPMs are stored as a set of linked lists of pointers to “chunks” in the DRAM, as shown in Figure 1. The multiply phase computes multiplications of all combinations of fetched elements, resulting in maximum reuse of inputs without any index matching, thus circumventing the problem of unproductive loads. Since each PE traverses through the non-zero elements of a row in Matrix $B$, the memory access during this phase is sequential and predictable. In addition, multiple PEs operate on the same row for each column element that corresponds to the row, resulting in high data reuse across the PEs.

In the merge phase, each M4 core is assigned a pointer array of chunks that correspond to a single row of the result matrix $C$, as shown in Figure 2. When merging the different chunks, the M4 core needs to ensure that all the elements in the final row are ordered by their column index. To ensure this ordering, each M4 core maintains a sorting list. The sorting list only needs to be big enough to hold one element from every chunk that is being merged by this core. This is because all the chunks are ordered by their column indices when they are produced in the multiply phase. Once the first element of every chunk is inserted into the sorting list, the steady state involves writing out the smallest element to DRAM and fetching one element to be sorted.

The merge phase, as shown in Figure 4, is broken down into three steps: initialization, sorting list construction, and on-demand sorting.

Step 1. Initialization Each chunk is augmented with metadata that is used to keep track of the number of elements that have been fetched by the core. During initialization, the metadata of each chunk assigned to the core is written into the scratchpad memory.

Step 2. Sorting list construction The M4 core begins constructing the sorting list by inserting the head of every chunk into the list (Step 2a). The list is sorted again each time an element is pushed into the list, based on the column index. The core iterates over all of its assigned chunks, and so the list starts with first element of every chunk. As the M4 core inserts elements from the scratchpad memory into the sorting list, the M0 core fetches the next elements of the chunk into the new empty blocks (Step 2b).

Step 3. On-demand sorting The M4 core pops the smallest element of the list to be placed in the output buffer (Step 3a). The M4 core checks the chunk that this popped element originated from, fetches the next element in the chunk, and pushes it into the sorting list (Step 3b). The popped element is compared against the element that is currently in the output buffer. If the indices of the two elements match, the values of the two elements are summed. If the indices do not match, the element in the output buffer is written to memory as the first element of one row in the result Matrix $C$. The popped element then becomes the new element in the output buffer, and the next element is fetched from the chunk of the last popped element. This process is repeated until all the assigned chunks have been processed. As the M4 core consumes data from the scratchpad memory, the M0 core independently fetches the next data of each chunk onto the emptied blocks (Step 3c).

Unlike the multiply phase where most of the memory accesses are sequential and there is plenty of data sharing between different PEs, the data accesses of the merge phase are mostly irregular, with no shared data across the Arm core pairs. Each core is assigned a disjoint pool of chunks, so that each Arm core pair operates on independent memory space. The location of each memory load is determined by the element that was popped from the sorting list. Therefore, the memory access is highly irregular and difficult to predict. Because the two phases have such drastically different access patterns, we implemented a reconfigurable architecture that can tune its memory hierarchy based on the needs of each phase.

C. Scratchpad Prefetching

While all the core computation of the merge phase is handled by the M4 cores, each M4 is paired with an M0 core (Figure 3), which acts as a programmable prefetcher. The primary purpose of the M0 core is to fill the private scratchpad with the elements of the PPM rows, so that the M4 core can grab its data from the scratchpad instead of the memory.

The M0 starts fetching the head elements of the chunks at the initialization step. As shown in Figure 2, M0 begins fetching data once the metadata of a chunk has been registered into the scratchpad memory. This allows the M4 core to immediately proceed to the construction of its sorting list, without waiting on the memory. As the M4 core pushes a new element from the scratchpad into the sorting list, the M0 core loads the next element of the chunk into the evicted space, until all the elements have been consumed.

Due to the size of the local scratchpad, there is a limit to the number of chunks that can be held in the scratchpad. This also limits the length of the sorting list maintained by the M4, since the length of the sorting list is equal to the number of individual chunks being merged. When the total number of
chunks assigned to an Arm core pair exceeds the maximum length of the sorting list \((L)\), the PPM rows are divided into subgroups of \(L\) PPM rows. The merge phase is then performed in multiple passes, each one generating an intermediate result of \(L\) merged chunks. During each pass, the intermediate results are written out to a temporary space in memory. Once there is enough capacity to merge the remaining chunks as well as the intermediate results, the final merge pass produces a single, fully merged row of result matrix \(C\). These intermediate passes are expensive because the data needs to be stored in external memory, and read again during the final merge pass. To minimize the number of passes, \(L\) needs to be as high as possible. However, for the M0’s prefetching to be effective, each chunk needs to have sufficient number of elements that have been loaded ahead in the scratchpad. Therefore, there exists a trade-off between the number of PPM rows that is tracked during the merge phase, and the number of elements that can be prefetched into the scratchpad for each PPM row.

D. Sorting Algorithm

We explore two possible implementations for the sorting algorithm in the merge phase: linear sorting and priority queue sorting. Linear sorting is where the element to be inserted into list is compared one-by-one down the list, until a smaller element is found. When the new element is inserted into position, the rest of the list must also be shifted by one. Because pushing and popping can both happen at the same time, linear sorting has \(O(L)\) complexity.

In priority queue sorting, the list is organized as a binary tree, where the root is always the smallest element of the list. When an element is inserted, the element is appended to the end of the tree. The tree is then re-balanced by recursively swapping the parent node and child node, if the child is smaller than the parent. When the smallest element (root) is popped, the root is replaced by a leaf node, and then the whole tree is re-balanced. The re-balancing process only has a complexity of \(O(\log(L))\), because the push and pop must happen independently as two separate steps. While the priority queue has better scalability as \(L\) increases, the high overhead of managing the binary tree favors linear sorting when \(L\) is small.

III. Circuit Implementation

Our chip consists of two compute substrates, as shown in Figure 5. The first, composed of 32 PEs (4 PEs/tile), computes the multiply phase. Each PE has a 32-bit Floating-Point (FP) multiplier and supports out-of-order loads/stores. The second substrate consists of eight ARM Cortex M0+M4 pairs (1 pair/tile) for the merge phase.

All the compute elements are connected through a reconfigurable network. The network consists of a fully-synthesizable Swizzle-Switch Network (SSN) crossbar based on [23], with the original pull-down networks replaced by OR trees (Figure 6). The synthesizable SSN still uses the same priority algorithm, but can also be easily ported to different process technologies since it does not require a custom layout. The crossbars support request coalescing, multicasting (Figure 8) and Least-Recently Granted (LRG) arbitration (Figure 7).

A. Compute Units

The PEs are custom Finite State Machine-based elements that perform the multiply phase of the outer product algorithm. At the core of the PE is a Control Unit (CU) that walks through the algorithm state machine. The CU initiates loads of elements of columns of Matrix \(A\) and rows of Matrix \(B\), tracking requests in a request queue. The request queue is a structure that allows out-of-order loads to the elements of the input matrices. Load responses satisfy an entry in the queue by associatively searching the address field of each request queue entry. Each PE also houses a single-cycle, single-precision floating point multiplier that multiplies elements of \(A\) and \(B\) as soon as they are available in the request queue. The calculated partial product elements are stored into a “data” store buffer. This is a simple FIFO queue of (address, data, valid) tuples. There exists a separate buffer to store pointers, which is associatively-searchable, unlike the data buffer. Through this split store buffer design, we are able to reduce the energy consumed by limiting expensive associative searches to fewer
registers. Finally, a debug block is used to relay important messages at programmable intervals to the off-chip interface, such as state of each PE, number of multiplications committed, etc.

The general purpose cores, Arm Cortex-M0 and Cortex-M4 cores, handle the computation in the merge phase. They are both low-power, in-order cores designed for high energy efficiency. The M4 performs the bulk of the computation including the floating-point operations. The M0 acts as a programmable prefetcher for loading data into the scratchpad independent of the M4’s operation.

The M0 and M4 cores communicate through the use of local scratchpad memory for shared data, and hardware mutex locks to streamline synchronization. The mutex locks come in two types: first-come-first-serve (FCFS) mutex and sleep mutex. The FCFS mutex is a simple synchronization lock where the core that acquires the lock first prevents the other core from acquiring the lock, until the first one releases it. When querying the lock for acquisition, the cores have the option to stall until the lock is freed. The sleep mutex is a unidirectional lock with a predetermined owner. Sleep mutex begins with its lock pre-acquired by its designated core, and the non-designated core stalls whenever it accesses a locked mutex. During the merge phase, the sleep mutex is used by the M4 core to prevent M0 core from starting the prefetch before M4 has finished initiating the metadata.

B. Coalescing Crossbar

The crossbar takes one cycle to arbitrate, based on a least-recently granted (LRG) scheme, and another cycle to transmit data. As shown in Figure 7, each requester sends its priority bits to be bitwise OR’d. The corresponding bit of the result vector, based on the index of the requesters, is sent back to the requesters and the one with a 0 on its granted bitline wins. Next cycle, the winner clears its priority bits and other requesters set the priority bit corresponding to the winner to 1, granting them higher priority than the winner. In any particular cycle, one column will always be zero among all requesters, since there will always be one with the highest priority. If any channel is not actively requesting, it will assert all 0s instead of its actual priority bits to put it on the lowest priority possible. For example in Figure 8, in Cycle 0, only requesters 1 and 2 request the channel, and therefore only these two assert their priority bits while 0 and 3 assert all zeroes. The result of the bitwise OR would be 1100, and then each requester checks their corresponding bit, in which case requester 2 wins. Since requesters 0 and 3 did not request, it ignores the result of the bitwise OR. The winner, in this case requester 2, then clears its priority bits. Once granted, the requester can hold on to the channel until it chooses to free the channel. Requests can be coalesced in the crossbar, shown in Figure 8. Since the channel can observe all the requesters and their requesting addresses, it can simply compare them with the winner’s address and grant to any matching requesters. Coalescence does not affect the priority status, since it happens after arbitration.
A. Frequency and Bandwidth Sweep

Figure 10 shows the clock and bandwidth sweeps for a matrix of size 100K×100K and density of 0.0008%. The multiply and the merge phases were evaluated separately in order to determine the optimal parameters for each phase. Clock sweeps show that while multiply performance hits a roofline, merge performance saturates slowly, as merge is more compute-heavy due to the overhead of maintaining the sorting list. We observe the frequency and voltage level in which the chip achieves optimal energy efficiency to be at 41.7 MHz and 0.860 V for the multiply phase, and 352.0 MHz and 0.864 V for the merge phase.

For the bandwidth sweeps, simulation results are appended to measured results to illustrate the impact of higher bandwidth and more compute units. The performance of multiply phase continues to increase with higher bandwidth, while the merge phase reaches saturation early, at less than 1 GB/s. The “knee”

C. Reconfigurable Cache

The downstream L0 crossbar connects to the reconfigurable L0 cache consisting of four logical SRAM banks, each of which consists of four physical SRAM banks. The L0 cache provides second-level coalescing by comparing the new requests with existing pending requests stored in the miss status holding registers (MSHR). Along with tracking missed requests, the MSHRs also act as a request queue that takes in the inbound requests, a fill buffer that temporarily holds the returned data before storing to SRAM and a response queue that sends the read data back to the PEs. For coalescence, each MSHR entry stores a bit vector of all requesters and adds additional requesters, should any coalesce in the process. The upstream crossbar then multi-casts the read data back to the PEs based on the requester bit vector.

For the multiply phase, the L0 is a multi-banked set-associative cache, allowing NZEs of B to be shared. For merge, it is reconfigured into a multi-banked scratchpad by disabling the tag array and the Least-Recently-Used (LRU) counter, and is private to each M0-M4 pair. Through another set of coalescing crossbars, the L0 cache in each tile connects to the L1 layer, which interfaces to the front side bus (FSB).

Only minor modifications were made to the cache controller to enable reconfiguration into scratchpad mode. In the scratchpad mode, the tag arrays and the set index bits are disabled, and the controller addresses directly into each SRAM bank.

IV. MEASURED RESULTS

The performance of our 2.0 mm×2.6 mm accelerator for Sparse Matrix-Matrix multiplication, with the chip layout shown in Figure 9, was evaluated through matrix squaring on synthetically matrices, as well as power-law graphs that are representative of real-world sparse matrices [24], [25]. The measured characteristics of the chip are summarized in Table I.

| Technology          | 40 nm CMOS                  |
|---------------------|-----------------------------|
| Die Size            | 3.0 mm × 3.0 mm             |
| Block Size          | 2.0 mm × 2.0 mm             |
| Total SRAM          | 112 Kb                      |
| Data Precision      | Single-Precision Floating Point |
| Nominal Frequency   | 41.7 MHz 0.860 V (Multiply) |
| (Minimum Energy)    | 352.0 MHz 0.864 V (Merge)   |
| Maximum Frequency   | 950.0 MHz 1.27 V            |
| Nominal Power       | 66.6 mW (Multiply)          |
| Consumption         | 226.0 mW (Merge)            |

Fig. 8: Crossbar and cache coalescence. The crossbar coalesces identical requests by marking the requesters in a bit vector, which is then stored in the cache controller. While it is in the cache controller, more requests can be coalesced along the way so that there be any requesters asking for the same address.

Fig. 9: Annotated 3.0 mm × 3.0 mm die photo with GDS overlay. There are eight tiles per chip, each tile containing an ARM Cortex-M0, a Cortex-M4, and four PEs.


lines show that the multiply phase is \( \sim 30 \times \) more sensitive to bandwidth than the merge phase.

Based on the frequency and bandwidth scaling of the chip, scaling out our current chip to \( 16 \times \) the current configuration would meet the CPU’s performance at \( 9.5 \times \) less bandwidth, \( 16.7 \times \) lower power and \( 0.08 \times \) the area. At this configuration, the chip will be able to make optimal use of available bandwidth by minimizing off-chip traffic.

\[ \text{B. Benefits of Reconfigurable Memory} \]

One of the key design choices of the chip is the use of reconfigurable memory that transitions between cache and scratchpad based on the demands of the algorithm. For workloads with well-defined data access and reuse patterns, the scratchpad improves performance over the cache by preventing any data that will be reused by the program from getting evicted out to memory during intermediate computation, ensuring each critical data to only be fetched once. Figure 11 shows the benefit of using the scratchpad memory during the merge phase at varying matrix densities, but with the matrix dimension fixed. We observe an average performance benefit of \( 25.7\% \) across the different matrices, with higher benefits for denser matrices.

\[ \text{C. Sorting Algorithm Comparison} \]

To determine the most suitable sorting algorithm for the merge phase, Figure 13 shows the performance comparison of linear sorting and priority queue sorting for scratchpad and no-scratchpad configurations, with (linear sorting, no-scratchpad) as the baseline. The linear sorting scheme outperforms the priority queue for both scratchpad and no-scratchpad. When the scratchpad is present, linear sorting exhibits as much as
21.8% improvement over the priority queue sorting, and the advantage increases for larger and denser matrices. This is because the maximum sorting list size of 16 is not large enough for the $O(\log(L))$ complexity of priority queue resulting in better scalability for larger lists. Based on the empirical results, the linear sorting algorithm is selected as the default sorting algorithm for the merge phase for this chip.

As discussed in Section II-C, the number of elements that can be prefetched into the scratchpad per chunk (i.e. block size) has an inverse relationship with the maximum length of the sorting list. A larger block size allows more elements of PPM row to be loaded in advance, but results in a smaller sorting list, which increases the likelihood of performing multiple merge passes. Figure 14 shows the performance impact of different block sizes. For large matrices of dimension greater than 1,000, block size of 4 with sorting list length of 16 outperforms block size of 2 and 8 by 4.7% and 7.0%, respectively. Therefore, the block size of 4 was chosen as the default configuration for the chip.

**D. Comparison with State-of-the-Art Approaches**

Figure 12 compares the energy and bandwidth efficiency of the chip executing sparse matrix-matrix multiplication against the highly-optimized, commercial software libraries on a high-end CPU (Intel Core i7) and GPU (Tesla V100). The matrix dimension, density, and pattern of non-zeros were varied to observe how different platforms react to each matrix parameter. For matrices with a uniformly-random distribution of non-zeros, the chip exhibits greater bandwidth efficiency for larger and denser matrices for both the CPU and GPU. In contrast, the improvement in energy efficiency over the CPU is more prominent when the matrix is small and sparse, but relatively constant against the GPU at any matrix size or density. This is because the performance of CPU degrades more prominently as density is lowered as amount of extraneous computation increases. On the other hand, GPU performance is relatively consistent because work is scheduled in large batches, and thus less sensitive to changes in data.

In the case of power-law graphs, as shown in Figure 15, the improvement in bandwidth efficiency exhibits a slight decrease with increasing NNZ for the GPU. The power-law graphs were synthetically generated using the Graph500 R-MAT data generator [27] to emulate the characteristics of real-world graph datasets.

Table II summarizes the key metrics of this work compared with that of the CPU, the GPU, a DSP [21], and an
TABLE II: Key metrics and comparison vs. CPU/GPU and prior work.

| Feature                              | Intel Core i7-6700K (MKL) | NVIDIA Tesla V100 (CUSP) | DSP+ [21] | ASIC+ [22] | This work |
|--------------------------------------|----------------------------|--------------------------|-----------|------------|-----------|
| Kernel                               | SpMM                       | SpMM                     | SpMV†     | DMM‡†     | SpMM      |
| Maximum Matrix Dimension             | 120,000                    | 120,000                  | 217,918   | 256        | 120,000   |
| Minimum Matrix Density               | 0.002%                     | 0.002%                   | 0.003%    | 3%         | 0.002%    |
| Reconfigurability                    | ×                          | ×                        | ✓         | ✓          | ✓         |
| Process (nm)                         | 14                         | 12                       | 40        | 14         | 40        |
| Core Count                           | 8                          | 5120                     | 4         | 16         | 48        |
| Total Core Area (mm²)                | 122.00                     | 815.00                   | 0.93      | 0.02       | 5.20      |
| Frequency (MHz)                      | 4000                       | 1250                     | 515       | 800        | 744‡‡     |
| Off-Chip Memory Bandwidth (GB/s)     | 34.10                      | 900.00                   | 3.20–8.53 | N/A (on-chip only) | 0.24     |
| Power (W)                            | 58.84                      | 123.95                   | 0.06      | 0.04       | 0.25††    |
| Compute Density (NNZs/mm²) [× 10⁶]  | 0.0279                     | 0.0129                   | 9.0183    | N/A        | 0.4775‡   |
| Energy Efficiency (NNZ/J) [× 10⁶]    | 0.58††                     | 0.87‡‡                   | 129.95††  | N/A        | 7.28††‡   |
| Bandwidth Efficiency (NNZ/GB) [× 10⁶]| 1.00                       | 0.15                     | 0.98–2.61 | N/A        | 11.73††‡  |

* Not directly comparable to this work † Sparse Matrix-Vector Multi ‡ Dense Matrix-Matrix Multi ‡‡ Area normalized to 40 nm technology †† 744 MHz for bandwidth efficiency. Multiply phase at 41.7 MHz and merge phase at 352 MHz were used for energy efficiency and power. ‡‡ Only on-chip energy ‡‡‡ Combines on-chip and off-chip energy. The GPU cores accounts for approx. 75% of the total combined energy [26].

ASIC [22]. The DSP is designed specifically for sparse matrix-vector multiplication, and the ASIC focuses on multiplication between matrices with relatively higher density (≥ 3%) using only on-chip storage. Therefore, these two works cannot be directly compared to our work. Our work built the first chip that aims at accelerating sparse matrix-matrix multiplication for real-world sized sparse matrices and addresses the off-chip memory bottleneck. The chip consumes 0.25 W on average when operating at its optimal energy efficiency point of 41.7 MHz for multiply and 352 MHz for merge. In general, our chip achieves an average energy efficiency gain of 12.6× against the CPU and 8.4× against the GPU. The compute density of the chip, which is throughputs (NNZ/s) per area, is 17.1× that of the CPU, and 37.1× that of the GPU. The bandwidth efficiency is a key metric measuring the number of non-zero elements in the result matrix computed per bandwidth used; it shows how well the accelerator can make use of the available bandwidth. This work is able to achieve 11.7× and 77.6× improvements in terms of bandwidth efficiency compared to the CPU and the GPU, respectively.

V. CONCLUSION

This paper presented a sparse matrix-matrix multiplication accelerator that leverages the outer product method of matrix multiplication to minimize redundant memory accesses. The 48 heterogeneous cores comprised of 32 custom PEs and 8 Arm Cortex M0+M4 pairs are tightly coupled via a coalescing crossbar and reconfigurable memory. The ability to switch from cache to scratchpad memory in different phases of the workload resulted in speedups of up to 27.3%. Our solution achieves an energy efficiency of 7.3 M output NNZ/J, and bandwidth efficiency of 11.7 M output NNZ/GB. This energy efficiency is 12.6× and 8.4× higher than that achieved by state-of-the-art software libraries on the CPU and GPU, respectively. Moreover, our solution achieves improvements of 11.7× and 77.6× compared to the CPU and GPU, in terms of bandwidth efficiency, which is the key figure-of-merit for memory-bound workloads such as SpMM.

ACKNOWLEDGEMENT

The material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7864. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.
REFERENCES

[1] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, “Outerspace: An outer product based sparse matrix multiplication accelerator,” in 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), Feb 2018, pp. 724–736.

[2] J. R. Gilbert, S. Reinhardt, and V. B. Shah, “A Unified Framework for Numerical and Combinatorial Computing,” Computing in Science & Engineering, vol. 10, no. 2, pp. 20–25.

[3] ———, “High-performance Graph Algorithms from Parallel Sparse Matrices,” Proc. of the Int’l Workshop on Applied Parallel Computing, 2006.

[4] A. Buluç and J. R. Gilbert, “The Combinatorial BLAS: Design implementation and applications,” The Int’l Journal of High Performance Computing Applications.

[5] V. B. Shah, “An Interactive System for Combinatorial Scientific Computing with an Emphasis on Programmer Productivity,” Ph.D. dissertation, June 2007.

[6] S. van Dongen, “Graph Clustering by Flow Simulation,” Ph.D. dissertation, 2000.

[7] A. Azad, A. Buluç, and J. R. Gilbert, “Parallel Triangle Counting and Enumeration Using Matrix Algebra,” 2015 IEEE Int’l Parallel and Distributed Processing Symposium Workshop, pp. 804–811, 2015.

[8] H. Kaplan, M. Sharir, and E. Verbin, “Colored intersection searching via sparse rectangular matrix multiplication,” SCG ’06: Proceedings of the twenty-second annual symposium on computational geometry, pp. 52–60, 2006.

[9] G. Penn, “Efficient transitive closure of sparse matrices over closed semirings,” Theoretical Computer Science, vol. 354, no. 1, pp. 72–81, 2006.

[10] V. Haplà, D. Horák, and M. Merta, Use of Direct Solvers in TFETI Massively Parallel Implementation. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 192–205.

[11] Y. Saad, Iterative methods for sparse linear systems. Siam, 2003, vol. 82.

[12] A. Buluç and J. R. Gilbert, “Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments,” SIAM Journal on Scientific Computing, vol. 34, no. 4, pp. C170–C191, 2012.

[13] A. Buluç and J. R. Gilbert, “Challenges and advances in parallel sparse matrix-matrix multiplication,” in 2008 37th International Conference on Parallel Processing. IEEE, 2008, pp. 503–510.

[14] E. Saule, K. Kaya, and Ü. V. Çatalyürek, “Performance evaluation of sparse matrix multiplication kernels on intel xeon phi,” in International Conference on Parallel Processing and Applied Mathematics. Springer, 2013, pp. 559–570.

[15] F. V’quez, G. Ortega, J. Fernández, I. García, and E. M. Garzón, “Fast sparse matrix matrix product based on ellr-t and gpu computing,” in 2012 IEEE 10th International Symposium on Parallel and Distributed Processing Workshops and Ph.D. IEEE, 2012, pp. 669–674.

[16] G. P. Krawezik and G. Poole, “Accelerating the ansys direct sparse solver with gpus,” in Symposium on Application Accelerators in High Performance Computing. SAAHPC, 2009.

[17] S. Dalton, L. Olson, and N. Bell, “Optimizing sparse matrixmatrix multiplication for the gnu,” ACM Transactions on Mathematical Software (TOMS), vol. 41, no. 4, p. 25, 2015.

[18] C. Y. Lin, H. K.-H. So, and P. H. Leong, “A model for matrix multiplication performance on fpgas,” in 2011 21st International Conference on Field Programmable Logic and Applications. IEEE, 2011, pp. 305–310.

[19] C. Y. Lin, N. Wong, and H. K.-H. So, “Design space exploration for sparse matrix-matrix multiplication on fpgas,” International Journal of Circuit Theory and Applications, vol. 41, no. 2, pp. 205–219, 2013.

[20] H. Giefers, P. Staar, C. Bekas, and C. Hagleitner, “Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of gnu, xeon phi and fpga,” in 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2016, pp. 46–56.

[21] R. Dorrance and D. Markovic, “A 190gflops/w dsp for energy-efficient sparse-blas in embedded iot,” in 2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits), June 2016, pp. 1–2.

[22] M. Anders, H. Kaul, S. Mathew, V. Suresh, S. Satpathy, A. Agarwal, S. Hsu, and R. Krishnamurthy, “2.9tops/w reconfigurable dense/sparse matrix-multiply accelerator with unified int8/int6/fp16 datapath in 14nm tri-gate cmos,” in 2018 IEEE Symposium on VLSI Circuits, June 2018, pp. 39–40.

[23] S. Satpathy, K. Sewell, T. Manville, Y. Chen, R. Dreslinski, D. Sylvester, T. Mudge, and D. Blaauw, “A 4.5tb/s 3.4tb/s/w 6464 switch fabric with self-updating least-recently-granted priority and quality-of-service arbitration in 45nm cmos,” in 2012 IEEE International Solid-State Circuits Conference, Feb 2012, pp. 478–480.

[24] A. Chapanon, M. S. Krishnamoorthy, and B. Yener, “Graph Theoretic and Spectral Analysis of Enron Email Data,” Computational & Mathematical Organization Theory, vol. 11, no. 3, pp. 265–281, Oct 2005.

[25] N. Satish, N. Sundaram, M. M. A. Puttawy, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, “Navigating the maze of graph analytics frameworks using massive graph datasets,” in Proceedings of the 2014 ACM SIGMOD Int’l conference on Management of data. ACM, 2014, pp. 979–990.

[26] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, “Power modeling for gpu architectures using mcpat,” ACM Trans. Design Autom. Electron. Syst., vol. 19, no. 3, pp. 26:1–26:24, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2611758

[27] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, “Introducing the graph 500,” 2010.