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

Dong-Hyeon Park, *Student Member, IEEE*, Subhankar Pal, *Student Member, IEEE*, Siying Feng, *Student Member, IEEE*, Paul Gao, *Student Member, IEEE*, Jielun Tan, *Student Member, IEEE*, Austin Rovinski, *Student Member, IEEE*, Shaolin Xie, *Member, IEEE*, Chun Zhao, *Senior Member, IEEE*, Aporva Amarnath, *Student Member, IEEE*, Timothy Wesley, Jonathan Beaumont, Kuan-Yu Chen, *Student Member, IEEE*, Chaitali Chakrabarti, *Fellow, IEEE*, Michael Bedford Taylor, *Member, IEEE*, Trevor Mudge, *Fellow, IEEE*, David Blaauw, *Fellow, IEEE*, Hun-Seok Kim, *Member, IEEE*, and Ronald G. Dreslinski, *Member, IEEE*

***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 consists of dedicated floating-point multiplication units, and general-purpose Arm Cortex-M0 and Cortex-M4 cores. The on-chip memory reconfigures 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*— Decoupled access execution, reconfigurablility and accelerator, sparse matrix multiplier, synthesizable crossbar.**

I. Introduction

THE emergence of big data and massive social networks has led to increased importance of graph analytics and machine learning workloads. One of the fundamental kernels that drive 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 toward the 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 (SpMM) 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 SpMM.

Since the number of zero elements in a sparse matrix largely outnumbers 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, and so on [11]. It represents an $N \times N$ sparse matrix using three arrays—values, column-indices, and row-pointers, with a total storage overhead of $2 \cdot \text{NNZ} + N + 1$ 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 a majority of computations are wasted on processing zero-value elements. Efficient SpMM that focuses only on the NZEs greatly improves 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

Manuscript received August 26, 2019; revised November 3, 2019; accepted December 5, 2019. Date of publication January 1, 2020; date of current version March 26, 2020. This article was approved by Guest Editor Ken Takeuchi. This work was supported in part by the Air Force Research Laboratory (AFRL) and in part by the Defense Advanced Research Projects Agency (DARPA) under Grant FA8650-18-2-7864. *(Corresponding author: Dong-Hyeon Park.)*

D.-H. Park, S. Pal, S. Feng, J. Tan, A. Rovinski, A. Amarnath, J. Beaumont, T. Mudge, and R. G. Dreslinski are with the Department of Computer Science and Engineering, University of Michigan, Ann Arbor, MI 48109 USA (e-mail: dohypark@umich.edu).

P. Gao, S. Xie, C. Zhao, and M. B. Taylor are with the Department of Electrical Engineering, University of Washington, Seattle, WA 98115 USA.

T. Wesley, K.-Y. Chen, D. Blaauw, and H.-S. Kim are with the Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI 48109 USA.

C. Chakrabarti is with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85281 USA.

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSSC.2019.2960480

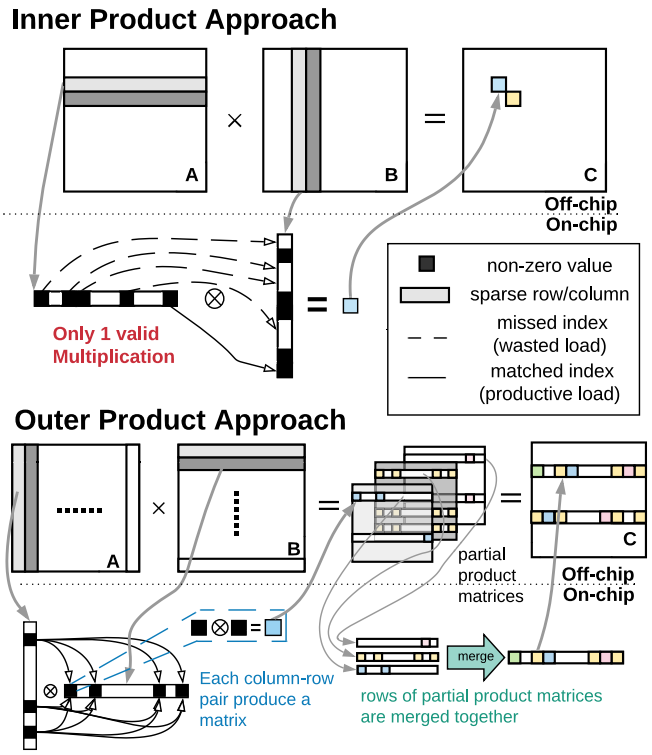


Fig. 1. High-level overview of SpMM using the inner-product and outer-product methods.

eliminating redundant memory accesses and maximizing data reuse.

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

A. Architectural Overview

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 the field-programmable gate arrays (FPGAs) [18], [19]. A comparative study of the 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 ($\geq$3%) matrix–matrix multiplication with small dimensions ($\leq$256) [22].

This article presents the first custom SpMM accelerator that addresses the off-chip memory access bottleneck for real-world-sized matrices, evaluating densities $\geq$0.002% and dimensions $\leq$120k. We evaluate our solution using the output NNZ per Joule (NNZ/J) for energy efficiency, which is equivalent to throughput per Watt. For bandwidth efficiency, we calculate the 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 a 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.

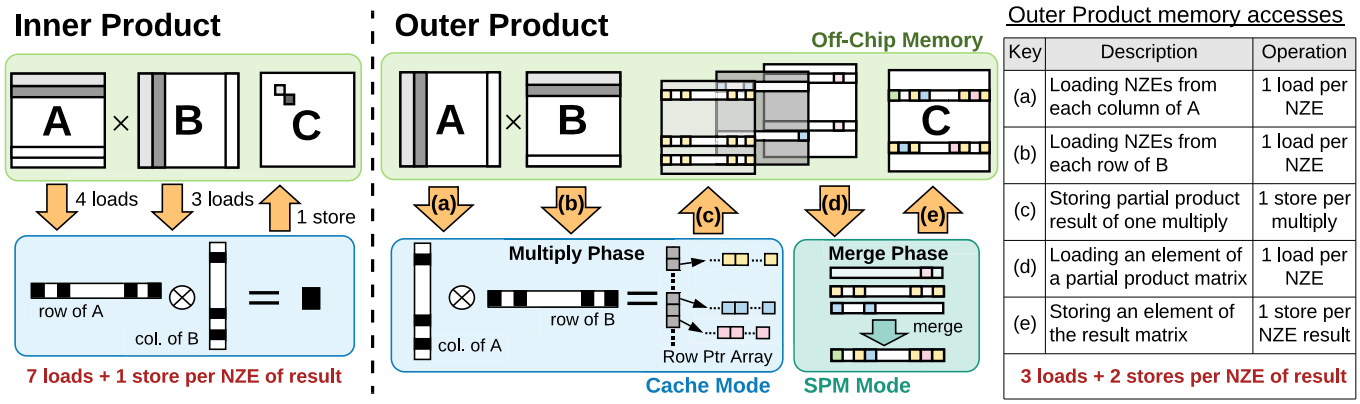
1) A custom architecture designed to take advantage of the unique data access pattern of the outer-product approach, where every memory fetch generates useful results.
2) A novel synthesizable coalescing crossbar that magnifies on-chip bandwidth.
3) Reconfigurable memory that can switch between the cache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

The rest of this article 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 SpMM [1], which performs $A \times B = C$, as shown in Fig. 2. The first operand, Matrix A is organized in the CSC format, while the second operand, Matrix B is in the 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.

We present a high-level description of our architecture in Fig. 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 [see Fig. 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 mergesort, and store the final results to DRAM. Our studies reveal that a scratchpad leads to better performance than a cache for this phase, due to the irregular nature of data accesses (see Section IV-B).



Outer Product memory accesses

Key	Description	Operation
(a)	Loading NZEs from each column of A	1 load per NZE
(b)	Loading NZEs from each row of B	1 load per NZE
(c)	Storing partial product result of one multiply	1 store per multiply
(d)	Loading an element of a partial product matrix	1 load per NZE
(e)	Storing an element of the result matrix	1 store per NZE result
3 loads + 2 stores per NZE of result		

Fig. 2. Breakdown of the off-chip memory accesses that are performed during the inner-product and outer-product methods. For the outer product, each NZE of the result matrix requires as little as three loads and two stores from off-chip memory, when there are no overlapping elements with the same indices during merge, which is the case for highly sparse matrices.

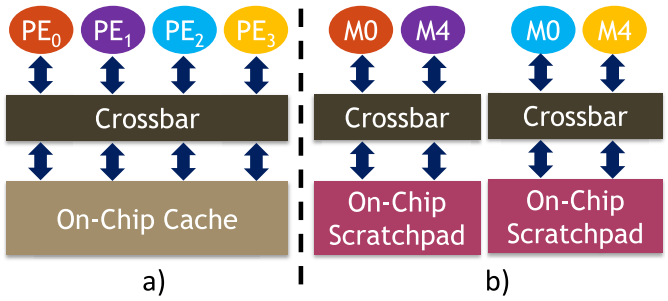


Fig. 3. High-level overview of the architectures suited for (a) multiply and (b) merge phases. Shared cache is suited for the predictable patterns in the multiply phase, whereas a private scratchpad is better for merge.

A. 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 PE multiplies an NZE of column i of A with all NZEs 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 Fig. 1. The multiply phase computes the 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 NZEs 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 Fig. 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 Fig. 4, is broken down into three steps: **initialization**, **sorting list construction**, and **on-demand sorting**.

Step 1 (Initialization): Each chunk is augmented with the 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 the 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

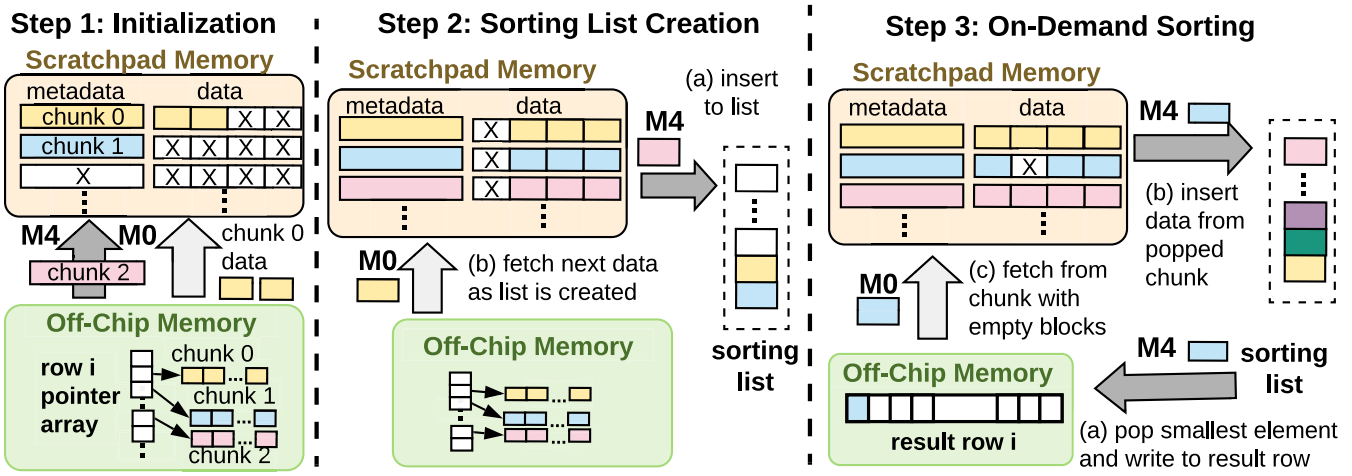


Fig. 4. Breakdown of the three steps of merge phase: initialization, sorting list construction, and on-demand sorting. M4 core performs the sorting operation on the data that have been loaded into the scratchpad by the M0 core.

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.

B. 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 (see Fig. 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 Fig. 2, M0 begins fetching data once the metadata of a chunk has been registered into the scratchpad memory. This allows the M4 core to proceed immediately 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 the result matrix C. These intermediate passes are expensive, because the data need 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 tradeoff between the number of PPM rows that are tracked during the merge phase and the number of elements that can be prefetched into the scratchpad for each PPM row.

C. 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 the 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 the 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 Fig. 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 the out-of-order loads/stores. The second substrate consists of eight ARM Cortex M0 + M4 pairs (1 pair/tile) for the merge phase.

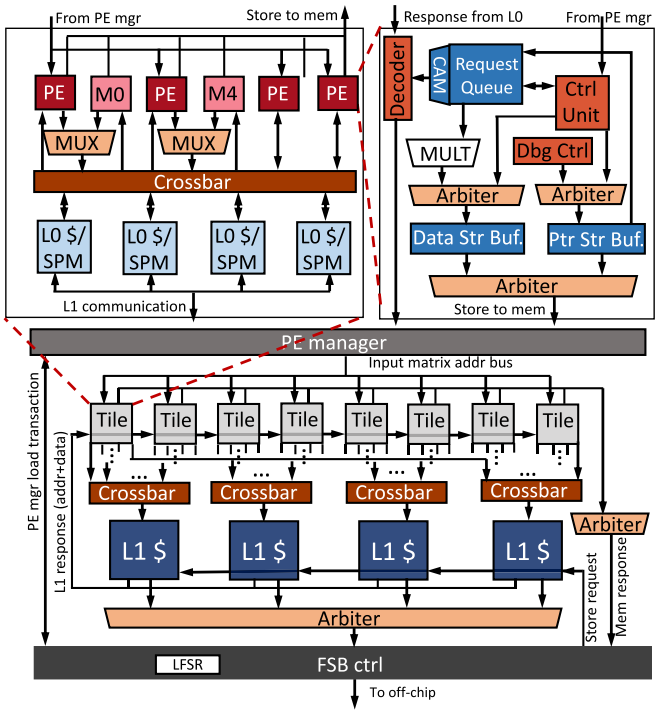


Fig. 5. Top-level diagram of the microarchitecture, a tile, and a PE. The chip contains a total of eight tiles, with each tile consisting of four PEs and a pair of M0 and M4 cores.

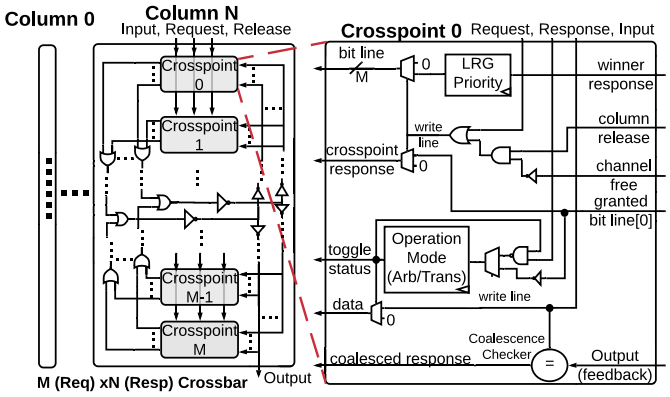


Fig. 6. OR trees of the SSN crossbar. Each crosspoint is one requester and the bitwise ORd results are sent back to each crosspoint.

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 (see Fig. 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 crossbar support request coalescing, least recently granted (LRG) arbitration (see Fig. 7), and multicasting (See Fig. 8).

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

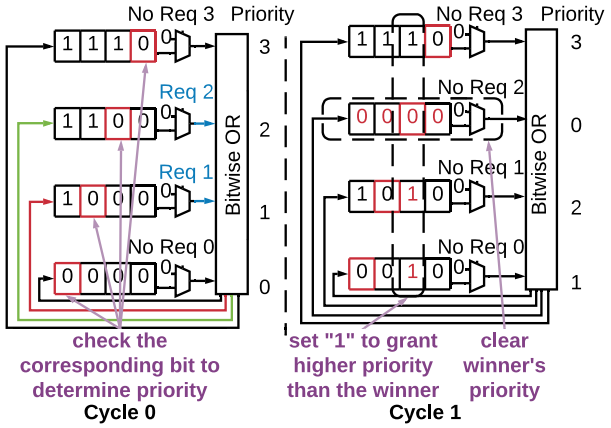


Fig. 7. LRG Scheme. The requesting nodes assert their priority bits and the winner is determined based on which requester receives a 0 in the corresponding response bit. The winner's priority bits are then cleared.

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 FP multiplier that multiplies the 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, and 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 the state of each PE, number of multiplications committed, and so on.

The general-purpose cores, Arm Cortex-M0 and Cortex-M4 cores, handle the computation in the merge phase. Both are low-power in-order cores designed for high energy efficiency. The M4 performs bulk of computation including the FP 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.

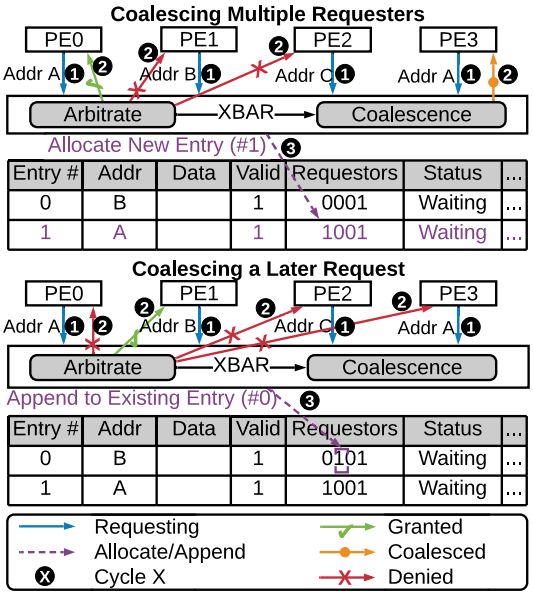


Fig. 8. Crossbar and cache coalescence. The crossbar coalesces identical requests by marking the requesters in a bit vector, which is then stored in the cache controller. While it is in the cache controller, more requests can be coalesced along the way should there be any requesters asking for the same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

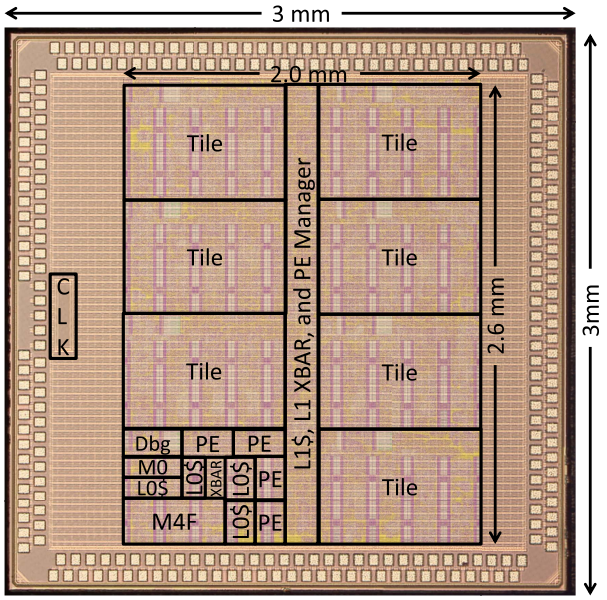


Fig. 9. Annotated 3.0-mm × 3.0-mm die photograph with GDS overlay. There are eight tiles per chip, each tile containing an ARM Cortex-M0, a Cortex-M4, and four PEs.

B. Coalescing Crossbar

The crossbar takes one cycle to arbitrate, based on an LRG scheme, and another cycle to transmit data. As shown in Fig. 7, each requester sends its priority bits to be bitwise ORd. 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. In the 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 Fig. 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 its 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, as shown in Fig. 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.

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 the 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. Each MSHR entry stores a bit vector of all requesters and adds additional requesters if there are any coalescence. 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 the 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 SpMM, with the chip layout shown in Fig. 9, was evaluated through matrix squaring on synthetic matrices, as well as power-law graphs that are the representatives of the real-world sparse matrices [24], [25]. The measured characteristics of the chip are summarized in Table I.

At the optimal frequency and voltage points, the accelerator achieves an energy efficiency of 6.1–8.4 M NNZ/J and a bandwidth efficiency of 6.4–15.5 M NNZ/GB. The SSN crossbar

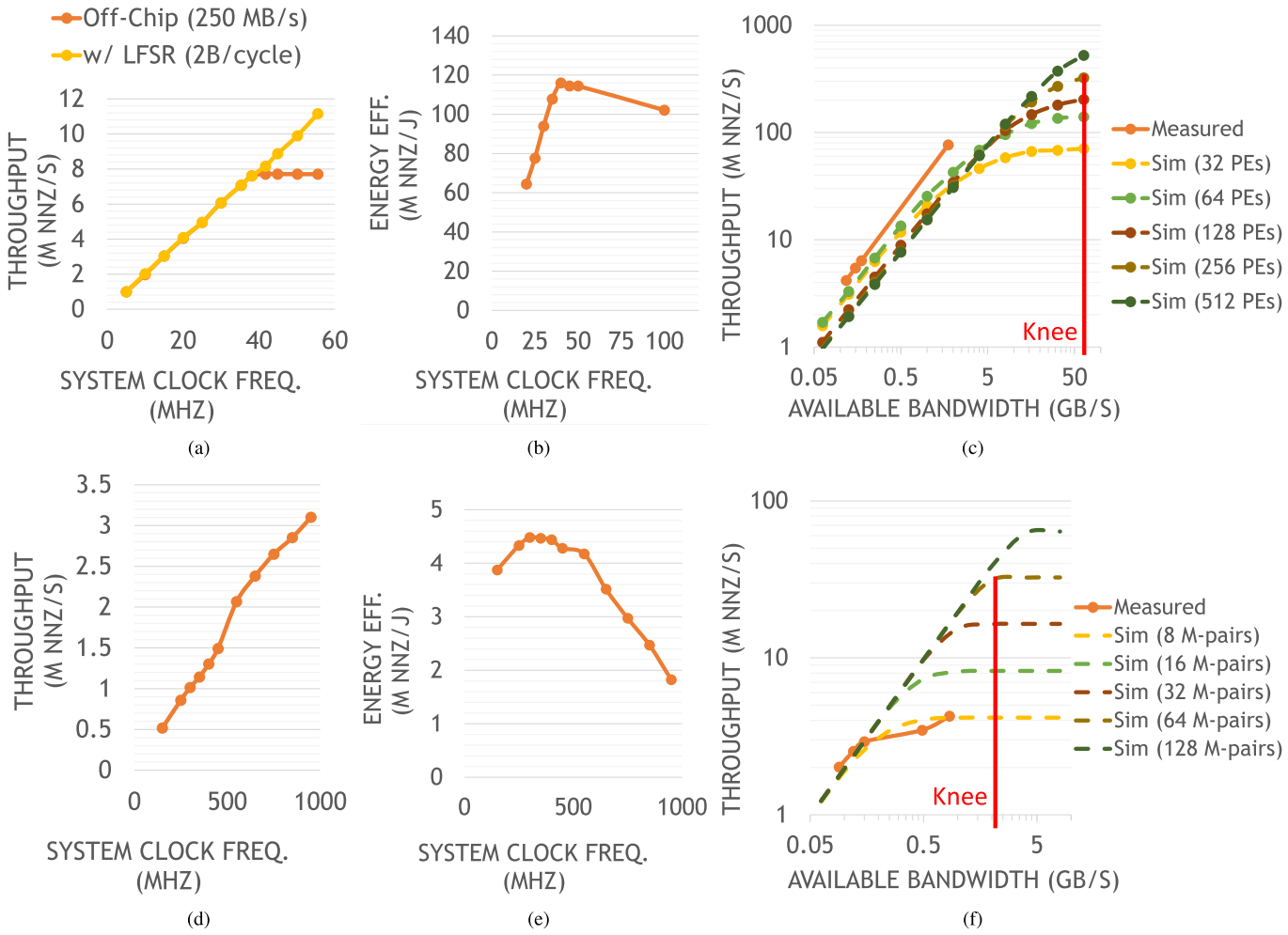


Fig. 10. Clock and bandwidth sweeps for matrix dim. 100k, density 0.0008%. For measurements with increased bandwidth, an on-chip LFSR is used for multiply and the M0 is used for merge. (a) Multiply throughput. (b) Multiply energy efficiency. (c) Multiply throughput bandwidth. (d) Merge throughput. (e) Merge energy efficiency. (f) Merge throughput bandwidth sweep.

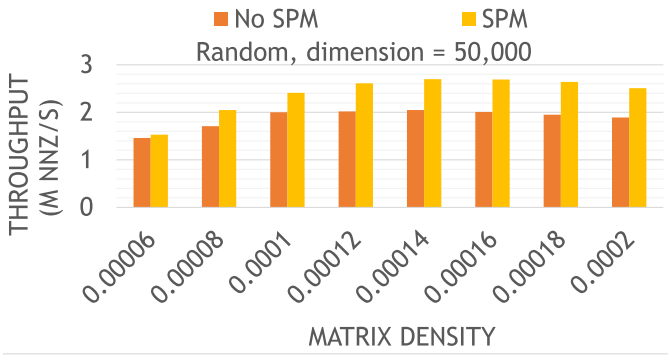


Fig. 11. Merge phase performance with and without scratchpad memory. Overall performance benefit of the scratchpad is 25.7%.

TABLE I
CHIP CHARACTERIZATION SUMMARY

Technology	40 nm CMOS
Die Size	3.0 mm × 3.0 mm
Block Size	2.0 mm × 2.6 mm
# Transistors	25,134,927
Total SRAM	112 KB
Data Precision	Single-Precision Floating Point
Nominal Frequency (Minimum Energy)	41.7 MHz 0.860 V (Multiply) 352.0 MHz 0.864 V (Merge)
Maximum Frequency	950.0 MHz 1.27 V
Nominal Power Consumption	66.6 mW (Multiply) 226.0 mW (Merge)

gives the chip a 24.9% performance gain at 86.3% the energy and 1.3% more area over an MUX crossbar-based design.

A. Frequency and Bandwidth Sweep

Fig. 10 shows the clock and bandwidth sweeps for a matrix of size 100k × 100k and a density of 0.0008%. The multiply and 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 the 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 the measured results to illustrate the impact of higher

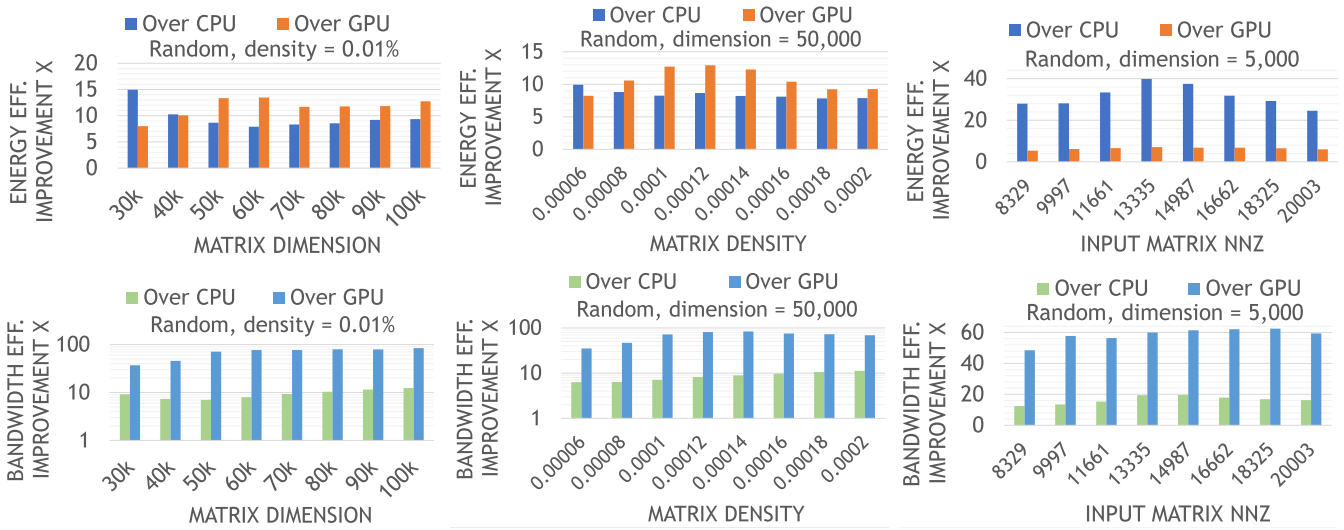


Fig. 12. Measured results over different matrices, showing energy and bandwidth efficiency of the proposed chip on uniform random matrices, normalized to a Core i7 CPU and V100 GPU running SpMM packages.

bandwidth and more compute units. The performance of the multiply phase continues to increase with higher bandwidth, while the merge phase reaches saturation early, at less than 1 GB/s. The "knee" lines show that the multiply phase is ∼30× 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× the current configuration would meet the CPU's performance at 9.5× less bandwidth, 16.7× lower power, and 0.08× the area. At this configuration, the chip will be able to make optimal use of available bandwidth by minimizing off-chip traffic.

B. Benefits of Reconfigurable Memory

One of the key design choices of the chip is the use of reconfigurable memory that transitions between the cache and the 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. Fig. 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.

C. Comparison With the State-of-the-Art Approaches

Fig. 12 compares the energy and bandwidth efficiency of the chip executing SpMM 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 the CPU degrades more prominently as density is lowered as the 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 the changes in data.

In the case of power-law graphs, as shown in Fig. 13, the improvement in bandwidth efficiency exhibits a slight decrease with the 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 the real-world graph data sets.

Table II summarizes the key metrics of this article compared with that of the CPU, the GPU, a DSP [21], and an application-specific integrated circuit (ASIC) [22]. The DSP is designed specifically for sparse matrix–vector multiplication, and the ASIC focuses on multiplication between the matrices with a relatively higher density (≥3%) using only on-chip storage. Therefore, these two works cannot be directly compared with our work. Our work built the first chip that aims at accelerating SpMM 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 throughput (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 NZEs in the result matrix computed per bandwidth used; it shows how well the accelerator can make use of the available bandwidth. This article is able to achieve 11.7× and 77.6× improvements in terms of bandwidth efficiency compared with the CPU and the GPU, respectively.

TABLE II
KEY METRICS AND COMPARISON VERSUS CPU/GPU AND PRIOR WORK

Feature \ Platform	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 (mm^2)	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 (NNZ/s/mm^2) [$\times 10^6$] **	0.0279	0.0129	9.0183	N/A	0.4775
Energy Efficiency (NNZ/J) [$\times 10^6$]	0.58††	0.87‡‡	129.95††	N/A	7.28††
Bandwidth Efficiency (NNZ/GB) [$\times 10^6$]	1.00	0.15	0.98–2.61	N/A	11.73^

* Not directly comparable to this work † Sparse Matrix-Vector Mult. ‡ Dense Matrix-Matrix Mult. ** Area normalized to 40 nm technology
** Number of Non-Zero-Elements per second (NNZ/s) is used in place of FLOPS to count only the operations that produce meaningful results.
^ 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]

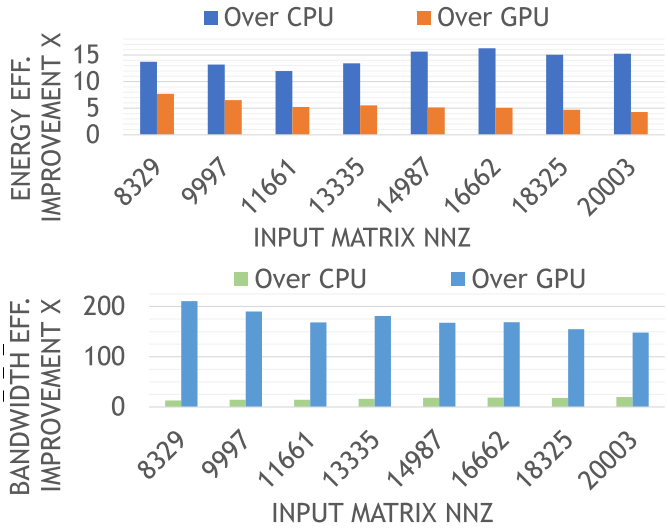


Fig. 13. Energy and bandwidth efficiency of the proposed chip on power-law graphs with a matrix dimension of 5000, normalized to a Core i7 CPU and V100 GPU running SpMM packages.

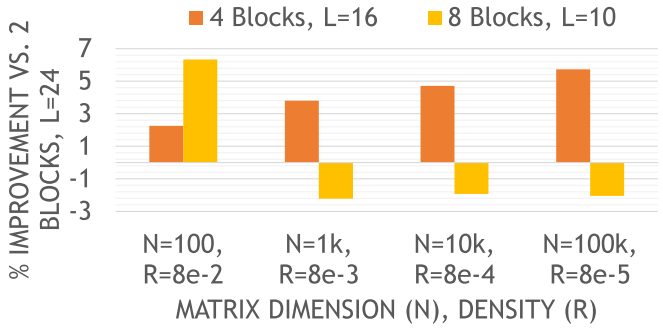


Fig. 14. Performance comparison between linear sorting and priority queue sorting at various matrix sizes and densities. The maximum size of the sorting list is 16.

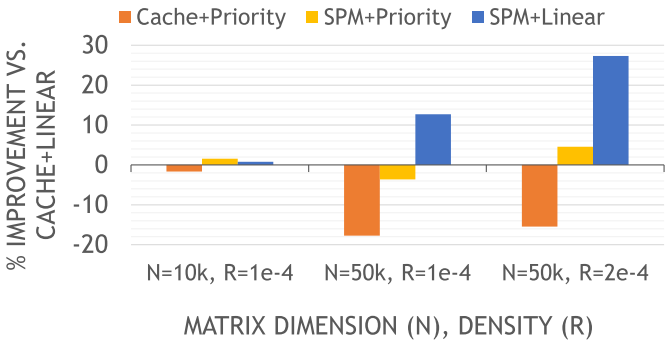


Fig. 15. Performance impact of various scratchpad block sizes and sorting list lengths for different matrix sizes and densities. Block size is the number of elements stored in the scratchpad per PPM row.

D. Sorting Algorithm Comparison

To determine the most suitable sorting algorithm for the merge phase, Fig. 14 shows the performance comparison of linear sorting and priority queue sorting for the 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-B, 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

the PPM row to be loaded in advance, but results in a smaller sorting list, which increases the likelihood of performing multiple merge passes. Fig. 15 shows the performance impact of different block sizes. For large matrices of dimension greater than 1000, block size of 4 with a sorting list length of 16 outperforms the 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.

V. Conclusion

This article presented an SpMM 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 a bandwidth efficiency of 11.7 M output NNZ/GB. This energy efficiency is 12.6× and 8.4× higher than that achieved by the state-of-the-art software libraries on the CPU and GPU, respectively. Moreover, our solution achieves improvements of 11.7× and 77.6× compared with the CPU and GPU, in terms of bandwidth efficiency, which is the key figure of merit for memory-bound workloads such as SpMM.

Acknowledgment

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 *et al.*, "OuterSPACE: An outer product based sparse matrix multiplication accelerator," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2018, pp. 724–736.

[2] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "A unified framework for numerical and combinatorial computing," *Comput. Sci. Eng.*, vol. 10, no. 2, pp. 20–25, Mar. 2008.

[3] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "High-performance graph algorithms from parallel sparse matrices," in *Proc. Int. Workshop Appl. Parallel Comput.*, 2006, pp. 260–269.

[4] A. Buluç and J. R. Gilbert, "The combinatorial BLAS: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, to be published.

[5] V. B. Shah, "An interactive system for combinatorial scientific computing with an emphasis on programmer productivity," Ph.D. dissertation, Dept. Comput. Sci., Univ. California, Santa Barbara, Santa Barbara, CA, USA, Jun. 2007.

[6] S. van Dongen, "Graph clustering by flow simulation," Ph.D. dissertation, Dept. Math., Utrecht Univ., Utrecht, The Netherlands, 2000.

[7] A. Azad, A. Buluc, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshop*, May 2015, pp. 804–811.

[8] H. Kaplan, M. Sharir, and E. Verbin, "Colored intersection searching via sparse rectangular matrix multiplication," in *Proc. 22nd Annu. Symp. Comput. Geometry (SCG)*, 2006, pp. 52–60.

[9] G. Penn, "Efficient transitive closure of sparse matrices over closed semirings," *Theor. Comput. Sci.*, vol. 354, no. 1, pp. 72–81, Mar. 2006.

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

[11] Y. Saad, *Iterative Methods for Sparse Linear Systems*, vol. 82. Philadelphia, PA, USA: SIAM, 2003.

[12] A. Buluç and J. R. Gilbert, "Parallel sparse matrix–matrix multiplication and indexing: Implementation and experiments," *SIAM J. Sci. Comput.*, vol. 34, no. 4, pp. C170–C191, Jan. 2012.

[13] A. Buluc and J. R. Gilbert, "Challenges and advances in parallel sparse matrix-matrix multiplication," in *Proc. 37th Int. Conf. Parallel Process.*, Sep. 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 *Proc. Int. Conf. Parallel Process. Appl. Math.* New York, NY, USA: Springer, 2013, pp. 559–570.

[15] F. V'zquez, G. Ortega, J. J. Fern'ndez, I. García, and E. M. Garzón, "Fast sparse matrix matrix product based on ELLR-T and GPU computing," in *Proc. IEEE 10th Int. Symp. Parallel Distrib. Process. Appl.*, Jul. 2012, pp. 669–674.

[16] G. Krawezik and G. Poole, "Accelerating the ANSYS direct sparse solver with GPUs," in *Proc. Symp. Appl. Accel. High Perform. Comput. (SAAHPC)*, 2009.

[17] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix—Matrix multiplication for the GPU," *ACM Trans. Math. Softw.*, vol. 41, no. 4, p. 25, 2015.

[18] C. Y. Lin, H. K.-H. So, and P. H. W. Leong, "A model for matrix multiplication performance on FPGAs," in *Proc. 21st Int. Conf. Field Program. Logic Appl.*, Sep. 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," *Int. J. Circuit Theory Appl.*, vol. 41, no. 2, pp. 205–219, Feb. 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 GPU, Xeon Phi and FPGA," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2016, pp. 46–56.

[21] R. Dorrance and D. Markovic, "A 190 GFLOPS/W DSP for energy-efficient sparse-BLAS in embedded IoT," in *Proc. IEEE Symp. VLSI Circuits (VLSI-Circuits)*, Jun. 2016, pp. 1–2.

[22] M. Anders *et al.*, "2.9 TOPS/W reconfigurable dense/sparse matrix–multiply accelerator with unified INT8/INTI6/FP16 datapath in 14 NM tri–gate CMOS," in *Proc. IEEE Symp. VLSI Circuits*, Jun. 2018, pp. 39–40.

[23] S. Satpathy *et al.*, "A 4.5 Tb/s 3.4Tb/s/W 64×64 switch fabric with self-updating least-recently-granted priority and quality-of-service arbitration in 45 nm CMOS," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2012, pp. 478–480.

[24] A. Chapanond, M. S. Krishnamoorthy, and B. Yener, "Graph theoretic and spectral analysis of Enron email data," *Comput. Math. Org. Theory*, vol. 11, no. 3, pp. 265–281, Oct. 2005.

[25] N. Satish *et al.*, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 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. Des. Autom. Electron. Syst.*, vol. 19, no. 3, pp. 26:1–26:24, Jun. 2014, doi: 10.1145/2611758.

[27] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group*, vol. 19, pp. 45–74, 2010.

Dong-Hyeon Park (Student Member, IEEE) received the B.S. degree in engineering from the Harvey Mudd College, Claremont, CA, USA, in 2014. He is currently pursuing the Ph.D. degree with the Computer Science and Engineering Department, University of Michigan, Ann Arbor, MI, USA, under the supervision of Prof. T. Mudge.

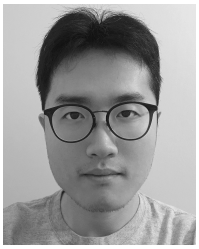
His research interests include sparse graph workloads and reconfigurable hardware architectures.

Subhankar Pal (Student Member, IEEE) received the B.E. degree in electrical and electronics engineering from the Birla Institute of Technology and Science at Pilani (BITS-Pilani), Pilani, India, in 2014, and the M.S. degree from the University of Michigan, Ann Arbor, MI, USA, in 2018, where he is currently pursuing the Ph.D. degree with the Computer Science and Engineering Department.

He was with NVIDIA, Santa Clara, CA, USA, where he was worked on pre-silicon verification and bring-up on multiple generations of GPUs. His research interests are in reconfigurable hardware accelerators and hardware-software co-design.

Siying Feng (Student Member, IEEE) received the B.S.E. degree in computer engineering from the University of Michigan, Ann Arbor, MI, USA, in 2017, and the B.S.E. degree in electrical and computer engineering from Shanghai Jiao Tong University, Shanghai, China, in 2017. She is currently pursuing the Ph.D. degree with the University of Michigan, under the supervision of Prof. R. Dreslinski.

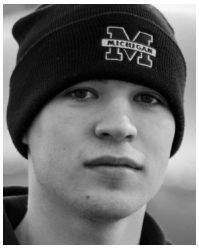
Her research interests include application specific accelerator design and reconfigurable hardware architecture.



Paul Gao (Student Member, IEEE) received B.E. degree from the Hong Kong University of Science and Technology, Hong Kong, in 2016, and the M.S. degree from the University of California at San Diego, San Diego, CA, USA, in 2018. He is currently pursuing the Ph.D. degree with the Bespoke Silicon Group, University of Washington, Seattle, WA, USA.

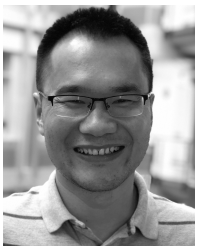
Jielun Tan (Student Member, IEEE) received the B.S.E. degree in electrical engineering from the University of Michigan, Ann Arbor, MI, USA, in 2017, where he is currently pursuing the M.S. degree under the supervision of Prof. R. Dreslinski.

His research interests include RISC-V related developments, massively parallel systems, and emerging technologies for computing.



Austin Rovinski (Student Member, IEEE) is currently pursuing the Ph.D. degree with the University of Michigan, Ann Arbor, MI, USA, under the supervision of R. Dreslinski from the Electrical Engineering and Computer Science Department.

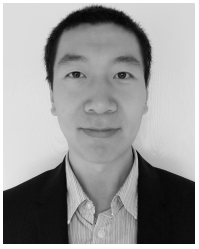
His current research interests include hardware accelerator design, and fast design, and implementation methodologies.



Shaolin Xie (Member, IEEE) received the B.S. and M.S. degree from Zhejiang University, Hangzhou, China, in 2004 and 2006, respectively, and the Ph.D. degree from the Chinese Academy of Science, Beijing, China, in 2009.

He was a Research Scientist with the Chinese Academy of Science and the Bespoke Silicon Group, University of Washington Division, Seattle, WA, USA. He is currently with Alibaba Group, Hangzhou.

His research interests include novel DSP manycore processor architecture.



Chun Zhao (Senior Member, IEEE) received the B.E. degree in electrical engineering from the Harbin Institute of Technology, Harbin, China, in 2007, and the Ph.D. degree in microelectronics from the University of Chinese Academy of Sciences, Beijing, China, in 2013.

He is currently a Research Associate with the Paul G. Allen School of Computer Science and Engineering, University of Washington, Seattle, WA, USA. His research interests include computer architecture, digital very-large-scale integration (VLSI) circuits, system on chip, and embedded systems, with a focus on the design and implementation of RISC-V-based microprocessors and accelerators.

Aporva Amarnath (Student Member, IEEE) received the B.E. degree in electrical and electronics engineering and the M.Sc. degree in economics from the Birla Institute of Technology and Science at Pilani (BITS-Pilani), Pilani, India, in 2015, and the M.S. degree from the University of Michigan, Ann Arbor, MI, USA, in 2017, where she is currently pursuing the Ph.D. degree with the Electrical and Computer Engineering Department.

Her research interests lie in working at the cusp of computer architecture and operating systems to build efficient and reliable systems using emerging technologies.

Timothy Wesley received the B.S. and M.S. degrees from the University of Michigan, Ann Arbor, MI, USA, in 2016 and 2019, respectively.

He is currently an Engineer working on AI accelerators at a start-up.

Jonathan Beaumont received the Ph.D. degree in computer science engineering from the University of Michigan, Ann Arbor, MI, USA, in 2019, under the supervision of T. Mudge.

He is currently a Lecturer with the University of Michigan. His research involves the design of high-throughput microarchitectures and how their parallelism can be made more accessible to programmers.

Kuan-Yu Chen (Student Member, IEEE) received the B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 2018. He is currently pursuing the Ph.D. degree with the University of Michigan, Ann Arbor, MI, USA.

His current research interests include digital circuit design, accelerators, and computer architecture.

Chaitali Chakrabarti (Fellow, IEEE) received the B.Tech. degree in electronics and electrical communication engineering from IIT Kharagpur, Kharagpur, India, in 1984 and the Ph.D. degree in electrical engineering from the University of Maryland, College Park, MD, USA, in 1990.

She is currently a Professor with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, USA. Her research interests include very-large-scale integration (VLSI) algorithm-architecture co-design of signal processing and communication systems and all aspects of low-power embedded systems design.

Michael Bedford Taylor (Member, IEEE) received the Ph.D. degree in electrical engineering and computer science from the Massachusetts Institute of Technology (MIT), Cambridge, MA, USA, in 2007.

He is currently a Professor with the Computer Science and Engineering and Electrical Engineering Department, University of Washington, Seattle, WA, USA. His research interests include application-specific integrated circuit (ASIC) clouds, Bitcoin mining hardware, dark silicon, tiled microprocessors, and open source hardware design.

Trevor Mudge (Fellow, IEEE) received the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 1977.

He is currently the Bredt Family Professor of computer science and engineering with the University of Michigan, Ann Arbor, MI, USA. He is the author of numerous articles on computer architecture, programming languages, very-large-scale integration (VLSI) design, and computer vision. He has chaired 56 theses in these areas.

Dr. Mudge received the ACM/IEEE CS Eckert-Mauchly Award in 2014 and the University of Illinois Distinguished Alumni Award.

David Blaauw (Fellow, IEEE) received the B.S. degree in physics and computer science from Duke University, Durham, NC, USA, in 1986, and the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 1991.

Since August 2001, he has been with the Faculty of the University of Michigan, Ann Arbor, MI, USA, where he is currently the Kensall D. Wise Collegiate Professor of Electrical Engineering and Computer Science Division and the Director of the Michigan Integrated Circuits Lab. He has performed extensive research in ultralow-power computing using subthreshold operation and analog circuits for millimeter sensor systems. For high-end servers, his research group introduced so-called near-threshold computing, which has become a common concept in semiconductor design. Most recently, he has pursued research in cognitive computing using analog, in-memory neural-networks for edge-devices, and genomics acceleration for precision health.

Hun-Seok Kim (Member, IEEE) received the B.S. degree in electrical engineering from Seoul National University, Seoul, South Korea, in 2001, and the Ph.D. degree in electrical engineering from the University of California at Los Angeles (UCLA), Los Angeles, CA, USA, in 2010.

He is currently an Assistant Professor with the University of Michigan, Ann Arbor, MI, USA. His research focuses on system analysis, novel algorithms, and very-large-scale integration (VLSI) architectures for low-power/high-performance wireless communications, signal processing, computer vision, and machine learning systems.

Ronald G. Dreslinski (Member, IEEE) received the B.S.E. degree in electrical engineering and computer engineering and the M.S.E. and Ph.D. degrees in computer science and engineering from the University of Michigan, Ann Arbor, MI, USA, in 2001, 2003, and 2011, respectively.

He is currently an Assistant professor of computer science and engineering with the University of Michigan. His research focuses on architectures that enable emerging low-power circuit techniques.