Sparse-T: Hardware accelerator thread for unstructured sparse data processing

Pranathi Vasireddy
University of North Texas
Denton, Texas, USA
pranathivasireddy@my.unt.edu

Krishna Kavi
University of North Texas
Denton, Texas, USA
krishna.kavi@unt.edu

Gayatri Mehta
University of North Texas
Denton, Texas, USA
gayatri.mehta@unt.edu

ABSTRACT

Sparse matrix-dense vector (*SpMV*) multiplication is inherent in most scientific, neural networks and machine learning algorithms. To efficiently exploit sparsity of data in the *SpMV* computations, several compressed data representations have been used. However, the compressed data representations of sparse date can result in overheads for locating nonzero values, requiring indirect memory accesses and increased instruction count and memory access delays. We call these translations of compressed representations as metadata processing. We propose a memory-side accelerator for metadata (or indexing) computations and supplying only the required nonzero values to the processor, additionally permitting an overlap of indexing with core computations on nonzero elements. In this contribution, we target our accelerator for low-end microcontrollers with very limited memory and processing capabilities. In this paper we will explore two dedicated ASIC designs of the proposed accelerator that handles the indexed memory accesses for compressed sparse row (CSR) format working alongside a simple RISC-like programmable core. One version of the the accelerator supplies only vector values corresponding to nonzero matrix values and the second version supplies both nonzero matrix and matching vector values for *SpMV* computations. Our experiments show speedups ranging between 1.3 and 2.1 times for *SpMV* for different levels of sparsities. Our accelerator also results in energy savings ranging between 15.8% and 52.7% over different matrix sizes, when compared to the baseline system with primary RISC-V core performing all computations. We use smaller synthetic matrices with different sparsities and larger real-world matrices with higher sparsities (below 1% non-zeros) in our experimental evaluations.

CCS CONCEPTS

• Hardware → Hardware accelerators; **• Computer systems organization → Embedded systems**; **Pipeline computing**.

KEYWORDS

Sparse matrix-dense vector multiplications, Compressed Sparse Row (CSR), Hardware accelerators, Application Specific Integrated Circuits, RISC-V

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549441

ACM Reference Format:
Pranathi Vasireddy, Krishna Kavi, and Gayatri Mehta. 2022. Sparse-T: Hardware accelerator thread for unstructured sparse data processing. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22), October 30-November 3, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3508352.3549441

1 INTRODUCTION

With the trend towards embedding intelligence into the *edge*, there is a growing need for architectural support for compute and storage-efficient machine learning algorithms on low-power sensing and handheld devices. These devices are characterized by simpler cores and small on-chip memories, often without cache memories [20, 21, 27]. Achieving real-time inference capability in these devices requires optimizing both the storage and computations performed. Matrix computations such as matrix-vector multiplication is an essential component of machine learning algorithms. For many practical applications, matrices contain a large proportion of zeroes whose storage and processing is wasteful. Hence sparsity (*the percentage of zeroes in the matrix*) can be exploited to improve performance, as well as reduce storage and energy requirements. [15, 22, 32]. Various sparse matrix representations have been proposed and used in scientific and machine learning codes. These include compressed sparse row (CSR [4]), block compressed CSR (BCSR [5]), compressed sparse column (CSC [6]), coordinate list (COO [10]), bit-vectors [22], and run-length encoding [22]. There are also some newer representations including hierarchical bit vectors [16] and compression on top of CSR [23]. Conceptually, compressed representations store only the nonzero (denoted NZ) values of a matrix along with *metadata* to identify the row and column positions (i.e., indices) of these values. Matrix codes are written to a specific sparse format in order to interpret the *metadata* and to perform computations only on the NZ values.

We claim that accessing and processing compressed *metadata* incurs overheads. For example, to perform pairwise multiplications of elements from matching columns locations *metadata* of one matrix is used to locate the nonzero elements of another. If the memory itself (or a small processing unit placed close to memory) could perform this *metadata* access and provide only needed nonzero values to the primary processing element, it saves the CPU energy and execution cycles. Such a memory system can provide computation-memory parallelism by overlapping *metadata* accesses with CPU computation. In this contribution we describe the design and evaluation of such a dedicated (or ASIC) memory-side hardware accelerator called *Sparse-T*.

There are several studies that propose intelligent and programmable prefetching of data, particularly for applications that rely on irregular data structures including linked lists and sparse data representations such as CSR. For example, IMP [31] proposes hardware support for prefetching data items that involve indirect accesses such as *m[v[j]]*, which can represent accessing elements of a vector based on the location of nonzero values of matrix rows using CSR-based sparse matrices. We would like to point out that while our *Sparse-T* has the affect of prefetching data for processing, *Sparse-T* should not be considered merely as a prefetcher. In general IMP [31] and other prefetchers only aid in prefetching data to the processor while our *Sparse-T* can be programmed to supply *only needed* data, rather than prefetching all data to the core. For *SpMV* application, *Sparse-T* can be programmed to provide only matching nonzero values when both the matrix and the vector are sparse. Likewise, while there have been many prior studies in terms of *decoupling* or *off-loading* memory access operation (consider an early decoupling work reported in [26]), *Sparse-T* is a flexible hardware which can be programmed to process application specific metadata processing. Helper threads (particularly software threads) have been used to aid primary threads with some operations (for example see [17]). Such software techniques may not lead to performance gains if the threads are scheduled on different cores requiring cache coherency related overheads. We use separate hardware unit specifically for indexing operations, placed near memory and hence eliminate cache coherency issues and compiler optimization that leads to performance loss in software threads.

This paper makes the following contributions.

(1) We designed two versions of ASIC memory-side accelerator, called *Sparse-T*: in the first case (*Sparse-T_1*), *Sparse-T* only provides vector values corresponding to nonzero matrix values (and the primary CPU core obtains nonzero matrix values); in the second case (*Sparse-T_2*), *Sparse-T* provides both matrix and vector values to the primary core, eliminating the need for the primary core accessing memory.
(2) Using ARM current standard technology cell libraries, we reported power, performance and area (PPA) for both the ASIC designs of *Sparse-T* using Synopsys design tool suite for *SpMV* computations.
(3) We evaluated performance gains (speedup and energy savings) with smaller synthetic matrices by varying sparsity levels and also presented results using real-world large sparse matrices with higher sparsities. We presented a comparison of the performance with RISC-V alone and RISC-V with *Sparse-T* designs.

In this work, our focus is on computations on low-end compute platforms. These microcontroller-based devices (MCUs) comprise simple in-order cores (such as a core from ARM Cortex-M series or RISC-V RV32) integrated with a small on-chip SRAM, clocked at no more than a few hundred MHz. Thus, achieving intelligence at the edge requires highly optimized implementations of various types of ML inference algorithms. However, the use of a memory-side accelerator such as our *Sparse-T* can be explored for other types of processing environments. The primary concerns of *SpMV* operation involve memory access latency, bandwidth utilization and parallelism. In the MCU integration, *BE* and *FE* units of *Sparse-T* issue requests directly to the on-chip SRAM via an on-chip interconnect. Thus bandwidth utilization is not a problem for *Sparse-T*. Similarly, *Sparse-T* is envisioned as an accelerator alongside single in-order CPU core. In case of multicore systems, it may be possible to use multiple *Sparse-T* accelerators, one per core. To summarize, *Sparse-T* proposed in this work focuses on decreasing memory access latency by overlapping computation time in primary RISC core with that of memory fetching in *Sparse-T* accelerator.

2 BACKGROUND

Accessing and processing compressed metadata incurs overheads. To perform pairwise multiplication of elements from matching columns (of rows of one matrix), metadata of one matrix is used to locate (and often match) the nonzero elements of another matrix (or vector). Consider the *SpMV* algorithm that multiplies a sparse matrix M by a dense vector V to produce an output (dense) vector Y. Figure 1 shows a sample 3×3 matrix M of compressed sparse row (CSR) representation.

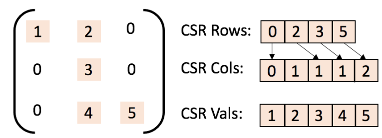

Figure 1: A 3x3 sparse matrix in CSR Format

In the CSR representation, a *cols* array holds the column indices of the nonzero values for each row of the matrix. A *rows* array holds pointers (indices) to the *cols* array where the row's non-zero column indices are stored. The *vals* array holds the NZ values. The *SpMV* algorithm traverses M row by row, obtains the column indices of the NZ values, and accesses the corresponding indices of the (dense) vector V. An outline of this algorithm implemented for a CSR representation of M is shown in Algorithm 1.

Algorithm 1 CSR Version of *spMV*

procedure SPMV(*M_rows*, *M_cols*, *M_vals*, n, v)
  $s \leftarrow 0$
  $k \leftarrow 0$
  for $i = 0; i < n; i = i + 1$ **do**
    $nnz \leftarrow M\_rows[i{+}1] - M\_rows[i]$
    $s \leftarrow 0$
    for $j = 0; j < nnz; j = j + 1$ **do**
      $s \leftarrow s + M\_vals[k{+}j] * v[M\_cols[k{+}j]]$
    $k \leftarrow k + nnz$
    $y[i] \leftarrow s$

Among the memory accesses made by this code, the indirect accesses performed by $v[cols[.]]$ are expensive – these indirect accesses require accessing $cols[.]$ before values of $v[.]$ can be read. As illustrated in the Algorithm 1, the Sparse Matrix-Vector multiplication (*SpMV*) is bottle-necked by memory accesses. Figure 2 illustrates this. Each iteration accesses are made to *M_cols* (*blue*

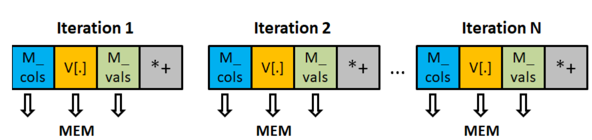

Figure 2: Metadata Overhead Memory Accesses in *SpMV*

blocks) to obtain non-zero column indices. Using these column indices, the values of V are read (shown in *yellow blocks*). Next, values from the sparse matrix row are read from $M\_vals[]$ (shown as *green blocks*). Finally, multiply-accumulate (MAC) operations are performed (shown as *gray blocks*) on values obtained from $M\_vals[]$ and $V[]$. There are 3 memory accesses per iteration per MAC operation.

We deem that fetching the elements of $M\_cols[]$ in order to access elements of $V[]$ as overhead – the CPU incurs the cost of fetching, decoding, and executing this memory access instruction (loading $M\_cols[]$) whose only usefulness is to provide the address for the memory access into array V. If the memory itself could perform this metadata access to fetch $V[]$, then it saves the CPU energy and cycles. Such a memory system can provide computation-memory parallelism by overlapping metadata accesses with CPU computation. This parallelism is depicted in Figure 2. Here, the memory system accesses the metadata first and performs a read of $V[.]$. The CPU no longer issues explicit metadata accesses followed by accesses to the vector V. Instead, the CPU directly reads the values of $V[]$ that the memory system has gathered. In this sense, the memory system can act as an accelerator to improve the overall performance of real-time ML code. Our memory-side *Sparse-T* accelerator, is motivated by this observation.

3 DESIGN OF *SPARSE-T*

Figure 3 shows the system organization of a typical embedded system. We envision our *Sparse-T* accelerator to be either embedded or placed very close to the RAM of a MCU as shown. In Figure 3, the black lines show that the primary CPU core still has access to both RAM and external flash memory of the device whereas the accelerator is connected only to the RAM. Our *Sparse-T* accelerator snoops over the memory requests sent from primary CPU core to memory over the memory bus to determine when to start fetching required data for CPU. In this section we describe the designs our proposed ASIC *Sparse-T* accelerator. We first describe (see Section 3.1) a design where *Sparse-T* supplies only the vector values corresponding to the nonzero values in the sparse matrix for the processor. In Section 3.2 we describe an ASIC design where *Sparse-T* supplies both the nonzero values of matrix and corresponding vector values to the processor. Both versions of *Sparse-T* use a 4-stage pipeline design and operate at the same clock rate as the main processor (Ibex [18] core).

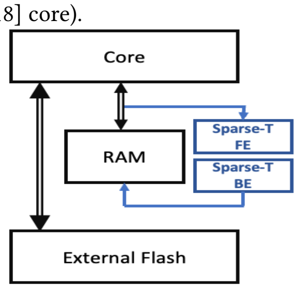

Figure 3: System Organization with *Sparse-T*

3.1 *Sparse-T* fetching vector values (Sparse-T_1)

In the first ASIC *Sparse-T* version, the 4-stage pipeline architecture is organized into memory buffers, front-end (*FE*) unit, back-end (*BE*) unit and processor-side buffer as shown in Figure 4. The *BE* loads matrix column indexes (location of nonzero matrix values) from CSR metadata and stream them to *FE* which fetches vector elements using the column indexes. The *FE* is responsible for CPU-side interactions, supplying vector elements to the CPU in response to buffer load requests.The architecture assumes at least two memory read ports; one for *FE* and one for *BE* to operate without stalls and in a decoupled manner synchronized by a control unit that starts or throttles the *BE* and *FE* units based on availability of space in the buffers.

3.1.1 Sparse-T *Front-End.*

The *Sparse-T FE* is responsible for fetching vector values using matrix metadata configuration, and coordinating with the CPU. The *FE* is supplied with matrix metadata by the primary processor core. This is achieved by writing to a set of memory-mapped registers (MMRs) upon initializing the *FE*. The MMRs needed of CSR-based *SpMV* multiplication are listed below.[1]

- M_Num_Rows: Number of rows of sparse matrix M.
- M_Rows_Base: Base address of CSR rows array of M.
- M_Cols_Base: Base address of CSR cols array of M.
- V_Base: Base address of dense vector V.
- $ElementSizes$: Sizes for Rows, Cols, Vals arrays and Vector.

For *SpMV* computations, *Sparse-T* provides *indexed gather* support. Values from vector $V[.]$ are gathered using indices from M_Cols to construct buffers. Vector values collected by *Sparse-T* are written into the scratch-pad memory shared between CPU and *Sparse-T* and are read by the CPU without load instructions. In our design, we assume a scalar load-store interface, but the *Sparse-T* design can work with vector load-store interfaces as well. The CPU obtains scalar or vector $v[.]$ values from *Sparse-T* and perform multiply-accumulate to produce the output vector. Whenever the CPU accesses the stored vector value, the *FE* updates its buffer state to determine when the buffer has been completely drained by the CPU. If multiple CPU-side buffers are available for *Sparse-T*, then whenever one buffer is drained, the *FE* switches to the next ready buffer. In this sense, the *FE* offers a streaming FIFO interface to the CPU. If the CPU performs a load when the buffer is not ready, then the *FE* stalls the load. Figure 4 describes the design of the *Sparse-T* pipeline operation.

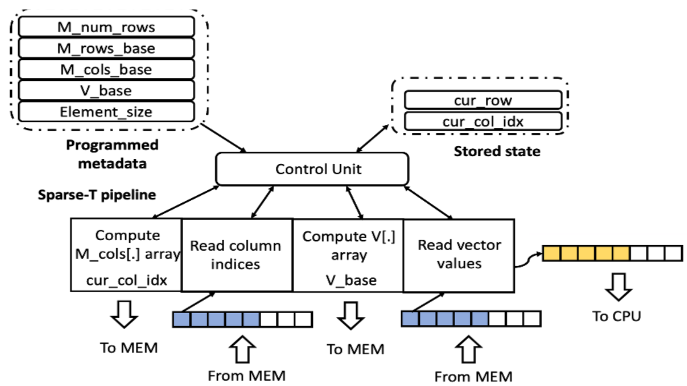

Figure 4: *Sparse-T* Pipeline fetching vector values

[1] We described ASIC *Sparse-T* for *SpMV* here. The design can be extended for Sparse matrix - Sparse vector *SpMSpV* multiplication using additional metadata and comparing indexes of Matrix columns with Vector indexes to match non-zero values.

3.1.2 Sparse-T *Back-End.* The *BE* pipeline stage issues memory read requests to obtain contents of the $M\_cols[.]$ array. Using the base address of $M\_cols[.]$ array (stored in register *M_cols_Base*) and element size *s*, the element address is calculated as M_cols_Base+s by *BE*. This computed address is used to generate requests to memory. The memory response obtained through the memory buffer is shared with *FE*. Column index values from *BE* are used to compute the addresses of the elements of array $V[.]$. This computed address is used to issue a second memory request in *FE* stage of the pipeline. Values read from array $V[.]$ are stored in a CPU-side buffer. While CPU issues load instruction for the matrix value fetch for the next operation, vector value for the operation is available from *Sparse-T*.

The control unit generates signals for all stages of the pipeline. In particular, the unit tracks processor buffer empty or full conditions to stall CPU load requests (when no ready buffer is available) or to stall memory request generation to $V[.]$ (when column indices have not yet been read from memory) or to skip issuing new memory read requests when all buffers are full. The unit also tracks the coordination between the pipeline stages and their communication with the memory buffers. Depending on the number of buffers provisioned to interface between *Sparse-T* and CPU, the control unit can also be configured to track which buffer to access.

3.2 *Sparse-T* fetching matrix and vector values (Sparse-T_2)

In the second version of *Sparse-T*, the *BE* calculates load address and fetches sparse matrix nonzero row and column indices from the memory system to enable the *FE* assemble CPU data buffer in a timely fashion. In addition, *FE* is responsible for fetching nonzero matrix and corresponding dense vector values for CPU and handling configuration writes from the CPU. In CSR representation, the difference of the current row index and previously fetched row index determines the number of non-zero values in the current row. Initially, the row address is calculated by incrementing the value stored in *M_Rows_Base* register by element size *s* and is stored in *cur_row* register in *BE*. The value in *cur_row* register will be used to calculate address for next elements by incrementing. At the start of each row, the current row index value is stored in *cur_row_idx* register. When switching rows, the *cur_row_idx* register value is mapped to *prev_row_idx* register while *cur_row_idx* register is updated with new value. To calculate the number of non-zeros in each row, the difference is calculated between *cur_row_idx* register value and *prev_row_idx* register value. This difference is stored in *cur_non_zero* register and is updated only when switching rows by *BE*. This value in *cur_non_zero* register determines the number of column indices to be fetched for that row. The column indices are fetched by calculating address as M_cols_Base+s and updating *cur_col_idx* register by *BE*. *FE* calculates the matrix values starting from *M_base* address by incrementing the value stored in *M_base* register with element size *s* and is stored in *cur_mat_val*. This value in *cur_mat_val* is incremented for each matrix value fetch. The vector address generation and value fetch by *FE* remain same as described in the previous architecture. Figure 5 describes the pipelined architecture of *Sparse-T*_2 design.

For this implementation, the *Sparse-T* constructs two values into the output buffer at each step – one is the nonzero values of the matrix row and the other is the vector value corresponding to this nonzero matrix value. The nonzero values of sparse matrix are pushed into the buffer first and vector value is pushed into the buffer next. CPU buffer is continuously updated with required nonzero matrix and vector value and there is a break only when moving the computation to the next row. Two memory buffers are still in use as described in previous architecture for *FE* and *BE* units to fetch memory values and the size of the buffers can remain as small as required. There are no additional metadata registers or pipeline stages required for the enhanced architecture. This architecture consumes slightly more area but less execution time compared to the *Sparse-T* architecture discussed in Section 3.1 and reduces the stalls due to memory access conflicts between CPU and *Sparse-T*. The matrix and vector values are available to the CPU directly from the scratch-pad memory without any additional load instructions.

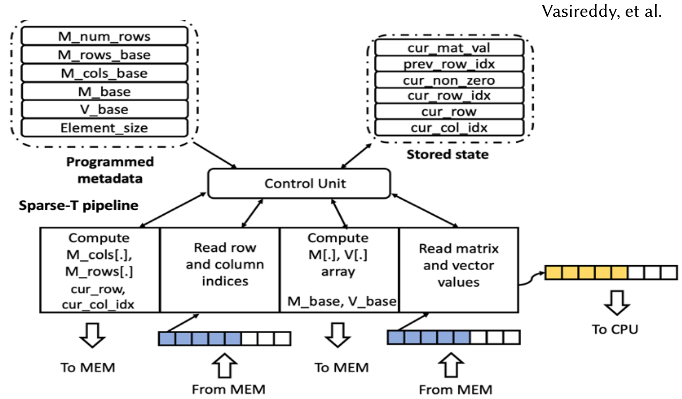

Figure 5: *Sparse-T* Pipeline fetching matrix and vector values

4 EXPERIMENTAL EVALUATION

In this paper we evaluated both ASIC *Sparse-T* designs described in the previous section (Section 3); one which fetches only the required vector values (*Sparse-T*_1) and the second fetches both matrix and vector values (*Sparse-T*_2) for *SpMV* operations in embedded processing environments.

System Configuration: Table 1 describes the system configuration used in our work. While evaluating the ASIC *Sparse-T* designs, we used Ibex RISC-V [18] core as the primary core. The ASIC hardware is designed using *Verilog* language to accurately model the embedded RISC-V Ibex core [18] and *Sparse-T* architectures. The system includes a 32-bit RISC-V [11] based instruction set architecture that supports compressed, integer multiply and divide, embedded and bit manipulation extensions also known as *Zero-Riscy* [18] along with required *Sparse-T* architecture. The primary CPU core uses an in-order 3-stage pipeline implementation. In particular, loads require two cycles to complete; hence stalling the pipeline for one cycle. ASIC *Sparse-T* is equipped with a 32-byte buffer to communicate with the primary CPU core. Both ASIC *Sparse-T* and CPU core evaluated at a maximum of 50MHz frequency.

Tools and Libraries Used: We used ARM standard libraries of 7nm, 16nm and 28nm to estimate the area occupied by *Sparse-T* architecture with respect to the CPU (Ibex [18]) core. However, the comparisons of ASIC *Sparse-T* with baseline (using only a Ibex RISCV core) for power, execution time and energy estimates are based on ARM 16nm technology node library with a standard threshold voltage (svt) of 80mV. We used *Synopsys Design Compiler* tool

Table 1: System Configuration

Processor	Values
Core	RISCV32 with IMAFDCV Extensions Frequency = 50 MHz In-order 3-stage Element Size (SEW) = 32 bit
ASIC *Sparse-T*	N=2 Buffers Buffer size = 32B

to generate gate-level netlist of the Verilog described hardware. Both the Verilog design and the synthesized netlist are verified for functionality against different workloads as specified below using Synopsys VCS tool. Synopsys Primetime is used to generate the power report for the netlist generated against the value change dump (VCD) trace file. We collected total execution cycles, area and power estimates for different combinations of RISC-V core and *Sparse-T* configurations.

Workloads: To analyze the performance of *Sparse-T*, synthetic matrices are generated. Since ML applications involve different sparsity levels, we generated synthetic matrices with different percentages of zero values (or sparsity) varying from 10% to 90% in steps of 10%. Experimental results are presented in Section 5 with synthetic sparse matrices of sizes 16*16, 32*32 and 64*64 for ASIC designs. We also included performance results for several matrices drawn from the Texas A&M Sparse Matrix collection (TAMU) [8]. These sparse matrices benchmark collection represent scientific workloads with very high levels of sparsities.

5 EXPERIMENTAL RESULTS

In this section we report power, performance and area for both ASIC *Sparse-T* designs and compare them with baseline of RISC-V alone. We used the same clock frequencies for the primary RISC-V core and *Sparse-T*. The two *Sparse-T* designs, (i) *Sparse-T* fetching vector values and processor fetching matrix values (*Sparse-T*_1 design), (ii) *Sparse-T* fetching matrix and vector values while processor only performs matrix-vector multiplications (*Sparse-T*_2 design) are compared with the baseline where the processor fetches matrix and vector values and performs the arithmetic computations.

5.1 Area Results

The area of ASIC *Sparse-T* is the sum of the logic gates of the control unit and storage for pipeline stages, *Sparse-T* buffers, memory-mapped registers, internal state registers and column-index storage. The area of *Sparse-T* varies by the type of variant chosen whereas processor area remains constant in all three configurations. The designs (*Sparse-T* and RISC-V) are synthesized using three of the recent semiconductor technology nodes (7nm, 16nm and 28nm of standard threshold voltage (svt) libraries from ARM) that are used in embedded devices. From Table 2, it can be observed that *Sparse-T*_1 design occupies 30.86% of the processor area while *Sparse-T*_2 design occupies almost 40.09% of the processor area for standard 16nm node technology. The reported area for *Sparse-T* designs considers RISC-V area as well since *Sparse-T* is an addition to the processor. *Sparse-T*_2 occupies more area compared to *Sparse-T*_1 since it requires more registers to store the state of the design as shown in Figure 5 and performs more work including calculating row address and addresses of nonzero matrix values; the latter is handled by CPU core in *Sparse-T*_1.

Table 2: Area in μm^2 of RISC-V and *Sparse-T* configurations

	Area(μm^2)		
Configuration	**7nm**	**16nm**	**28nm**
RISC-V alone	1451	5543	10961
RISC-V with *Sparse-T*_1	1920	7254	14651
RISC-V with *Sparse-T*_2	2059	7766	15664

Table 3: Power in μW on 16*16 matrix size at 10% sparsity for RISC-V and *Sparse-T* configurations

	Power in μW		
Configuration	**10MHz**	**25MHz**	**50MHz**
RISC-V alone	71	181	367
***Sparse-T*_1 alone**	18	44	88
RISC-V of *Sparse-T*_1	75	195	377
Total RISC-V with *Sparse-T*_1	93	239	465
***Sparse-T*_2 alone**	102	116	125
RISC-V of *Sparse-T*_2	63	168	323
Total RISC-V with *Sparse-T*_2	165	274	448

Table 4: Execution time in in μs on 16*16 matrix size at 10% sparsity for RISC-V and *Sparse-T* configurations

	Execution time in μs		
Configuration	**10MHz**	**25MHz**	**50MHz**
RISC-V alone	327	136	68
RISC-V with *Sparse-T*_1	225	90	41
RISC-V with *Sparse-T*_2	156	62	32

5.2 Power Results

The power estimates reported in Table 3 include both leakage power and dynamic switching power using 16nm technology process running at 10MHz, 25MHz and 50MHz frequencies for 16*16 matrix size. It is observed that *Sparse-T*_1 alone without RISC-V consumes less power compared to *Sparse-T*_2 without RISC-V at the reported frequencies. This additional power consumption in *Sparse-T*_2 design is because there is more switching activity than *Sparse-T*_1 design as it has to compute address for matrix values as well. However, it is also observed that RISC-V processor consumes less power when it is accelerated by *Sparse-T*_2 (represented as RISC-V of *Sparse-T*_2 power) compared to rest of the configurations (RISC-V of *Sparse-T*_1 and RISC-V alone). Due to the load balancing in RISC-V with *Sparse-T*_2 design, power consumption is also distributed among primary RISC-V core and *Sparse-T*_2. It is to be noted that RISC-V of *Sparse-T*_1 has higher power consumption compared to RISC-V of *Sparse-T*_2 and RISC-V alone since RISC-V in this configuration performs memory accesses for matrix values as well as access the buffered vector values supplied by *Sparse-T*. In the case of RISC-V alone, the RISC core is accessing both matrix and vector values, but it does not use the additional buffers as needed when a using *Sparse-T*, and the RISC-V core will be stalled during the memory accesses. The total power of RISC-V with *Sparse-T*_1 is given by sum of RISC-V of *Sparse-T*_1 and *Sparse-T*_1 alone in Table 3. Similarly, total power of RISC-V with *Sparse-T*_2 is sum of RISC-V of *Sparse-T*_2 and *Sparse-T*_2 alone.

Table 5: Energy in nJ for different matrix sizes at 10% sparsity and 50MHz frequency on RISC-V and *Sparse-T* configurations

	Energy (nJ)		
Configuration	**16*16**	**32*32**	**64*64**
RISC-V alone	24	94	372
RISC-V with *Sparse-T_1*	20	78	305
RISC-V with *Sparse-T_2*	12	45	176

5.3 Execution time Results

From the pipeline representations shown in Figure 4 and Figure 5, it can be observed that the execution time of the *Sparse-T* designs depend on the number of nonzero column index values. *Sparse-T*_1 supplies a new vector element at every clock cycle after the initial delay of 9 cycles to primary RISC-V core. *Sparse-T*_2 design supplies a new nonzero matrix value and a corresponding vector value every 3 cycles after an initial delay of 14 cycles. Hence the execution time of the *Sparse-T*_1 design is given as a product of the number of nonzero column indices in sparse matrix times the clock period added to the initial delay. However, Ibex processor [18] requires 3 cycles for a multiplication (MUL) and hence by the end of the execution on previous operands, both *Sparse-T*_1 and *Sparse-T*_2 designs will be able to supply new data to CPU without any CPU stalls waiting for *Sparse-T*. Also, if the processor has multiple buffers, then *Sparse-T* fetching both matrix and vector values will greatly reduce the loading of data and wait times compared to the other two configurations. Table 4 shows execution times that include arithmetic operations for matrix-vector multiplication and address computations (for 16*16 matrices at 10% sparsity). *Sparse-T*_2 design results in the lowest execution time, since *Sparse-T* handles the memory accessing and CPU performs multiply-accumulate operations in parallel. Whereas in *Sparse-T*_1 design, *Sparse-T* fetches vector values, while RISC-V fetches matrix values and perform computations which increase the total execution time of *Sparse-T*_1 design compared to *Sparse-T*_2 design with RISC-V. Although RISC-V in *Sparse-T*_2 design requires less execution times than *Sparse-T*_1 design, RISC-V with *Sparse-T*_2 requires additional 10% of hardware area to accommodate matrix value computations.

5.4 Energy Results

Energy consumption is particularly important for embedded computing devices. Since execution time is saved by using *Sparse-T* to support the processor with loading matrix and vector values, *Sparse-T*_1 and *Sparse-T*_2 designs result in energy savings compared to the RISC-V alone configuration as can be seen from Figure 6. However, the performance improvements depend on the amount of work offloaded to *Sparse-T*. The reported values are for both *Sparse-T* designs and RISC-V processor running at 50MHz frequency and synthesized using 16nm process technology. For *Sparse-T* designs, RISC-V is considered to be executing in parallel that adds to the area and power while reducing execution time. Since memory accesses in *Sparse-T* overlap with CPU's arithmetic operations, *Sparse-T* is never turned off during the entire execution time. Figure 6 shows that *Sparse-T*_1 design achieves between 15.8% and 5.4% energy savings for sparsities between 10% to 90% and *Sparse-T*_2 design achieves between 50.2% and 18.9% energy savings. On average, *Sparse-T*_1 and *Sparse-T*_2 achieve 13.7% and 38.7% energy savings respectively.

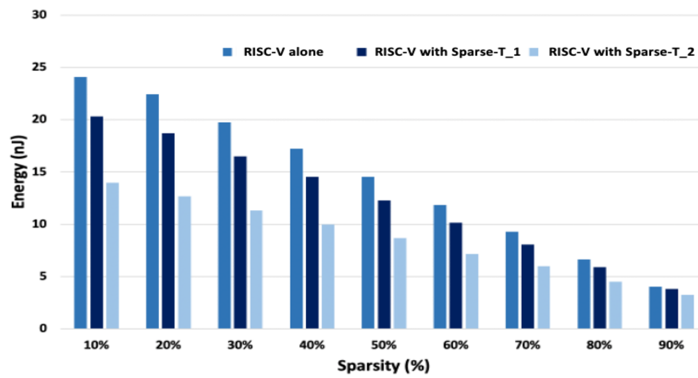

Figure 6: Energy in nJ for 16*16 matrix size over sparsity percentage

Using different matrix sizes of 16*16, 32*32 and 64*64 with 10% sparsity, power and execution times are obtained at 50MHz frequency to calculate the energy savings shown in Table 5. Energy savings slightly increase with increasing size of the matrix; 15.8% on 16*16, 17% on 32*32 and 18% on 64*64 for *Sparse-T*_1 and 50.2% on 16*16, 52.1% on 32*32 and 52.7% on 64*64 as the number of non-zero values at chosen 10% sparsity increase with matrix size. From the table, it can also be observed that both *Sparse-T*_1 and *Sparse-T*_2 perform better than RISC-V across all the matrix sizes due to the offloading and compute-memory overlap. In *Sparse-T*_2 design, *Sparse-T* executes almost half of the instructions (3 loads) required in each of the *SpMV* loop iteration and hence shows 50% reduction in energy compared to the baseline RISC-V alone performing both load and arithmetic operations. *Sparse-T*_1 design still requires RISC-V to compute arithmetic computations and hence shows lower savings.

5.5 Benchmark Evaluation

In most of the real world applications, large matrices are often used to store the information compared to the smaller matrix sizes presented in previous analysis. But for fast and effective parallel processing of these large matrices in machine learning and image processing applications, they are broken into smaller batches or blocks of size 16, 32 and 64. However, our *Sparse-T* accelerator can also perform well when dealing with large matrices. Table 6 shows the results for six large sparse matrices drawn from different domains of engineering from SuiteSparse Matrix Collection [8]. These scientific matrices exhibit very high degree sparsities. The distribution of nonzeros across the rows of the matrix (symmetry) does not have a large impact on the speedup and energy savings on our accelerator unlike the common prefetchers. Among the matrices selected, *jpwh*, *rbsa_480* and *pesa* have asymmetric nonzero distribution and the other three benchmarks (*685_bus*, *G10* and *Andrews*) are symmetric. *rbsa_480* and *G10* have 7.4% and 5.9% nonzero values and show higher speedups and energy savings with *Sparse-T* compared to other benchmarks with 1% nonzeros. Among the other benchmarks (*685_bus*, *jpwh*, *pesa* and *Andrews*), as matrix size increases, we notice an increase in the speedup and energy savings. The energy savings and speedups of RISC-V with *Sparse-T*_1 and RISC-V with *Sparse-T*_2 exhibit the same trend that was observed with smaller matrices. Since *Sparse-T*_2 has equal distribution of workload with RISC-V, it has more energy savings

Table 6: Percentage of energy savings and speedup for matrices from SuiteSparse Matrix collection [8] at 50MHz frequency on RISC-V and *Sparse-T* configurations

	Energy savings and Speedup						
Benchmark suite	**Matrix size**	**Number of non-zeros**	**Application**	**RISC-V with *Sparse-T*_1 Energy savings**	**RISC-V with *Sparse-T*_1 Speedup**	**RISC-V with *Sparse-T*_2 Energy savings**	**RISC-V with *Sparse-T*_2 Speedup**
685_bus	685 x 685	1,967	Power network problem	6.21%	1.38x	27.78%	1.79x
jpwh	991 x 991	6,027	Semiconductor device problem	10.78%	1.46x	38.80%	2.01x
rbsa480	480 x 480	17,088	Robotics problem	14.17%	1.53x	45.65%	2.26x
G10	800 x 800	38,352	Undirected weighted random graph	15.22%	1.54x	46.34%	2.28x
pesa	11738 x 11738	79,566	Directed weighted random graph	13.21%	1.47x	40.84%	2.02x
Andrews	60000 x 60000	410,077	Computational graphics/vision problem	14.63%	1.50x	44.49%	2.14x

and higher speedup when compared to *Sparse-T*_1 design where the workload is unequally distributed between primary core and accelerator (primarily doing more work than the accelerator).

6 RELATED WORKS

Sparse Matrix Accelerators. Accelerating sparse matrix operations has received attention from both the hardware and software communities. On the hardware side, works propose hardware acceleration of the entire computation: some of these works include a CAM-based accelerator [30], accelerator for very large *SpMV*. The work in [25] proposes a Two-Step *SpMV* algorithm and a memory-based accelerator to accelerate such computations on very large, very sparse graphs. Our work is different: we focus on reducing memory latency issues of embedded systems for matrix computations. Unlike works that aim to move the entire computation to a dedicated accelerator, our goal is simply to reduce the memory bottleneck faced by vectorized codes running on traditional cores. Some researchers explored hardware that expands sparse data into dense by inserting zeroes [3], [1]. It is believed that at lower sparsities, such expansion can improve performance since the expanded data can be executed using vector and SIMD instructions.

In [19] hardware SVM-based accelerator is designed which relies on data prefetchers and Compressed Sparse Column (CSC) format to reduce the number of indirect memory accesses and speed up *SpMV* computations. CSC format is similar to CSR format but compresses along columns. This allows for reuse of vector values by computing partial results using the nonzero values in each column of the matrix. These accelerators require (possibly floating point) multipliers inside the accelerators unlike our *Sparse-T* which only calculates memory addresses and requires only simple integer ALUs (possibly with shift operations instead of multipliers). The design reported in [28] takes advantage of the DRAM interleaving storage for improving bandwidth utilization in *SpMV* computations but our implementation focuses on embedded processors which do not have DRAMs and do not suffer from bandwidth utilization.

There are several works that focus on performance of sparse matrices for scientific applications. Authors of [7] proposed a parallel sparse matrix algorithm based on SUMMA used in BLAS library and parallelized the sparse matrix multiplication, while we used hardware accelerator to extract only nonzero values. Greathouse [14] proposed an algorithm, CSR-Stream to compute sparse Matrix - dense Vector multiplication for smaller rows. They also present a CSR-Adaptive algorithm which chooses CSR-Stream instead of traditional CSR, and expands sparse matrices to dense to enable parallelization. Azad and Buluc [2] proposed a parallel sparse Matrix - sparse Vector (*SpMSpV*) algorithm that stores the product of sparse Matrix - dense Vector based on the row indices and later accumulates it, all by using buckets.

Processing In Memory and Near Data Processing Approaches. There have been many studies on near-data processing (or Processing-In-Memory) approaches for improving memory latencies and utilize higher bandwidths. More recent works focused on migrating computations to PIM. Some older reports proposed migrating memory intensive operations closer to memory including memory allocation and garbage collection functions (see for example [9, 24, 29]). In one interesting work, the authors propose creating memory gestures (or macros) for some common operations involved in traversing linked lists and avoid bringing intermediate nodes into processor caches [12].

New Sparse Representations. In a different vein, there have been proposals on improving compression of sparse matrices and proposed techniques including hierarchical bit vectors [16] or compression on top of CSR [23]. There are proposals for specialized hardware to compress and decompress data for use by CPU (assuming that the CPU uses conventional *SpMV* software) [23]. Others propose hardware for new compression formats (such as hierarchical bit maps) for performing sparse matrix computations [16]. We programmed *Sparse-T* to handle sparse data represented using SMASH [16] format. SMASH format requires complicated indexing to locate the row and column positions of non-zero values of a sparse matrix. This implies that *Sparse-T* for SMASH is performing more work than the CPU, causing CPU to idle. Moreover, we feel that SMASH format may not be suitable for embedded systems. Due to space limitations, we did not include of the performance gains achieved when *Sparse-T* is programmed using Spike simulator [13] to process hierarchical bit representation of sparse data as done in SMASH [16].

7 CONCLUSIONS

In this work, we presented two ASIC designs of a memory-side accelerator for sparse matrix-dense vector multiplications. The accelerator, denoted as *Sparse-T*, decouples the overhead of accessing and interpreting sparsity metadata from the primary CPU core. Our approach should be distinguished from most other accelerators that accelerate the entire computation, not just index computations. In addition, we focus on micro-controller domain, necessitating low power design. We presented the ASIC implementation of *Sparse-T* that handles CSR sparse data representations. Although not shown in this paper, we have evaluated ASIC *Sparse-T* designs for other sparse representations like bit-vector and run-length. However, CSR format is chosen in this work since it is widely used. While more specialized sparse formats may be explored for specific domains and specific sparsity levels, they are likely require more complex programming and/or more complex hardware support.

The two ASIC *Sparse-T* designs presented in this paper show average performance gains between 1.3 and 2.1 depending on the sparsity levels with small synthetic and large real-world matrices over RISC-V baseline. The *Sparse-T* designs also result in energy savings, as high as 18% with *Sparse-T* fetching only vector values and 52.7% with *Sparse-T* fetching both matrix and vector values when compared to baseline with a RISC-V alone performing indexed computations for *SpMV* over different matrix sizes.

We are currently exploring the design of programmable *Sparse-T* using a bare minimum RISC-V like instructions with very few integer instructions, registers and caches so that different sparse representations and access patterns can be processed by *Sparse-T*.

Acknowledgements This research is supported in part by a SRC grant AIHW Task 2943 and a NSF grant #1828105. The authors acknowledge Nagendra Gulur (TI), Mahesh Mehendale (TI), Chris Tsongas (TI), Jaekyu Lee (ARM) as well as their UNT colleagues Charles Schelor, Shashank Adavally and Alex Weaver for their valuable suggestions.

REFERENCES

[1] Shashank Adavally, Nagendra Gulur, Krishna Kavi, Alex Weaver, Pranoy Dutta, and Benjamin Wang. 2020. ExPress: Simultaneously Achieving Storage, Execution and Energy Efficiencies in Moderately Sparse Matrix Computations. In *The International Symposium on Memory Systems*. 46–60.

[2] Ariful Azad and Aydin Buluç. 2017. A Work-Efficient Parallel Sparse Matrix-Sparse Vector Multiplication Algorithm. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 688–697. https://doi.org/10.1109/IPDPS.2017.76

[3] Adrián Barredo, Jonathan C Beard, and Miquel Moretó. 2019. Poster: Spidre: Accelerating sparse memory access patterns. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 483–484.

[4] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*. ACM.

[5] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, Calgary, Alberta, Canada, August 11-13, 2009*, Friedhelm Meyer auf der Heide and Michael A. Bender (Eds.). ACM, 233–244.

[6] Aydin Buluç, Samuel Williams, Leonid Oliker, and James Demmel. 2011. Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*. IEEE, 721–733.

[7] Aydin Buluç and John R. Gilbert. 2012. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. *SIAM J. Scientific Computing* 34 (2012).

[8] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1 (2011), 1:1–1:25. https://doi.org/10.1145/2049662.2049663

[9] S. Donahue, M.P. Hampton, R. Cytron, M. Franklin, and K.M. Kavi. 2002. Hardware support for fast and bounded time storage allocation. (May 2002).

[10] Aiyoub Farzaneh, Hossein Kheiri, and Mehdi Abbaspour. 2009. An efficient storage format for large sparse matrices. *Communications de la Faculté des Sciences de l'Université d'Ankara. Séries A1: Mathematics and Statistics* 58 (01 2009).

[11] RISCV Foundation. 2020. RISC-V: The Free and Open RISC Instruction Set Architecture. (2020). https://riscv.org

[12] L.M. Fox, C.R. Hill, R.K. Cytron, and K.M. Kavi. 2003. Optimization of storage-referencing gestures. (Oct. 29 2003).

[13] Abraham Gonzalez. 2019. The RISC-V ISA Simulator. (2019). https://chipyard.readthedocs.io/en/latest/Software/Spike.html

[14] Joseph L. Greathouse and Mayank Daga. 2014. Efficient Sparse Matrix-vector Multiplication on GPUs Using the CSR Storage Format. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New Orleans, Louisana) *(SC '14)*. IEEE Press, Piscataway, NJ, USA, 769–780. https://doi.org/10.1109/SC.2014.68

[15] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark Horowitz, and Bill Dally. 2016. Deep compression and EIE: Efficient inference engine on compressed deep neural network. In *2016 IEEE Hot Chips 28 Symposium (HCS), Cupertino, CA, USA, August 21-23, 2016*. IEEE, 1–6.

[16] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri-Ghiasi, Taha Shahroodi, Juan Gómez-Luna, and Onur Mutlu. 2019. SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 600–614.

[17] Jaejin Lee, Changhee Jung, Daeseob Lim, and Yan Solihin. 2008. Prefetching with helper threads for loosely coupled multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems* 20, 9 (2008), 1309–1324.

[18] lowRISC. 2017. Ibex: An embedded 32 bit RISC-V CPU core. (2017). https://www.ibex-core.readthedocs.io/en/latest/03_reference/pipeline_details.html

[19] Asit Mishra Nurvitadhi, Eriko and Debbie Marr. 2015. A sparse matrix vector multiply accelerator for support vector machine. *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)* (2015).

[20] NXP. 2019. Microprocesssors and Microcontrollers. (2019). https://www.st.com/en/microcontrollers-microprocessors.html

[21] NXP. 2019. NXP Microcontrollers Overview. (2019). https://www.nxp.com/docs/en/supporting-information/BL-Micro-NXP-Microcontroller-Overview-James-Huang.pdf

[22] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel S. Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 27–40.

[23] Arjun Rawal, Yuanwei Fang, and Andrew A. Chien. 2019. Programmable Acceleration for Sparse Matrices in a Data-Movement Limited World. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. IEEE, 47–56.

[24] M. Rezaei and K. Kavi. 2006. Intelligent memory manager: Reducing cache pollution due to memory management functions. *Elsevier Journal of Systems Architecture* 52, 2 (Jan 2006), 207–219.

[25] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C. Hoe, Larry T. Pileggi, and Franz Franchetti. 2019. Efficient SpMV Operation for Large and Highly Sparse Matrices using Scalable Multi-way Merge Parallelization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 347–358. https://doi.org/10.1145/3352460.3358330

[26] James E Smith. 1982. Decoupled access/execute computer architectures. *ACM SIGARCH Computer Architecture News* 10, 3 (1982), 112–119.

[27] TI. 2019. MSP432P401R, MSP432P401M SimpleLink Mixed-Signal Microcontrollers. (2019). http://www.ti.com/lit/ds/symlink/msp432p401r.pdf?ts=1587791677379

[28] Yaman Umuroglu and Magnus Jahre. 2014. An energy efficient column-major backend for FPGA SpMV accelerators. *2014 IEEE 32nd International Conference on Computer Design (ICCD)* (2014).

[29] W.Li, S. Mohanty, and K. Kavi. 2006. Page-based software-hardware co-design of a dynamic memory allocator. *IEEE Computer Architecture Letters* 5 (July 2006).

[30] Leonid Yavits and Ran Ginosar. 2017. Sparse Matrix Multiplication on CAM Based Accelerator. *CoRR* abs/1705.09937 (2017). arXiv:1705.09937 http://arxiv.org/abs/1705.09937

[31] Xiangyao Yu, Christopher Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*. IEEE, 178–190.

[32] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. 2017. Hello Edge: Keyword Spotting on Microcontrollers. *CoRR* abs/1711.07128 (2017). arXiv:1711.07128 http://arxiv.org/abs/1711.07128