ExPress: Simultaneously Achieving Storage, Execution and Energy Efficiencies in Moderately Sparse Matrix Computations

Shashank Adavally  
University of North Texas

Nagendra Gulur  
University of North Texas

Krishna Kavi  
University of North Texas

Alex Weaver  
University of North Texas

Pranoy Dutta  
University of North Texas

Benjamin Wang  
University of North Texas

ABSTRACT

Sparse matrix computations have witnessed a resurgence with the pervasive use of deep neural networks. Leveraging sparsity enables efficiency of storage by avoiding storing zeroes. However, sparse representations incur metadata computational overheads – software needs to process the metadata (or index) that describes row/column locations of non-zero values before it can access the corresponding data values. There have been several formats proposed for representing sparse matrices including Compressed Sparse Row (CSR), Coordinate (COO), Bitmaps, Run-length encoding, & hierarchical representations. Each representation achieves different levels of memory compression and incurs different levels of computational complexity depending on the sparsity (percentage of zero values). We seek answers to the following: (i) at what sparsity levels is it worth eliminating compressed representation of matrices and use the dense representation that includes both zeros and non-zero values, and (ii) even if we use compressed data representation, will it be useful to expand the matrices internally to eliminate metadata processing overheads? In this paper we propose the use of a special hardware called ExPress that expands compressed matrices into dense data, eliminating metadata computations from the main processing element. Our ExPress hardware is configurable so that it can expand from different compressed formats.

Our experiments for matrix-vector multiplication using several DNN workloads show performance gains of 43%, 33% and 11% on average over software implementations that use CSR, Bitmap and Run-length encoding respectively. ExPress shows performance gains over sparse software codes for sparsity up to 70%. Further, ExPress simultaneously achieves energy improvement by reducing the instruction overhead of sparsity-aware computations.

1 INTRODUCTION

With the trend towards embedding intelligence into the edge, there is a growing need to support 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 memory [36, 37, 46]. Achieving real-time inference capability in these devices requires optimizing both the storage and computations performed by matrix-based kernels such as matrix-vector multiplication. Leveraging sparsity (zeroes) in the input data and/or weights of deep neural nets (DNNs) has emerged as a viable technique to achieve these improvements [18, 39, 49].

Sparsity (the percentage of zeroes in the matrix) is exploited to improve performance, as well as reduce storage and energy requirements. To achieve these improvements, various sparse matrix representation techniques have been proposed and used in scientific and machine learning codes. These include compression formats such as compressed sparse row (CSR [2]), block compressed CSR (BCSR [3]), compressed sparse column (CSC [4]), coordinate list (COO [10]), bit-vectors [39], run-length encoding [39] and hierarchical bit-vector representations [27]. Conceptually, these formats store only non-zero (denoted NZ) values of a matrix along with metadata to indicate the row and column positions (i.e., indices) of these values. Matrix codes are written to a specific representation in order to interpret the metadata and to perform computations only on the NZ values.

We observe that accessing and processing compressed metadata incurs overheads. To perform pairwise multiplications of elements from matching columns (rows), metadata of one matrix is used to lookup (and often match) the non-zero elements of another. To illustrate, 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 $3 \times 3$ matrix $M$ and two compressed representations: compressed sparse rows (CSR) and Bitmap.

![Figure 1: A 3x3 sparse matrix in CSR and Bitmap Formats](https://example.com/fig1.png)
In the Bitmap representation, an array stores 0s and 1s corresponding to positions of zeroes and non-zeroes in the matrix. Only the non-zero values are stored in a separate array. Using the locations of the 1s in the bitmap array, software determines the corresponding column indices to perform matrix computations. For example, the \texttt{spMV} (sparse matrix-vector multiplication) algorithm traverses the bitmap array row by row, obtaining the column indices of the NZ values (corresponding to 1 bits in the bitmap), and accesses the corresponding indices of the (dense) vector \textit{V}. A simplified outline of this algorithm implemented is shown in Algorithm 1 below.

\begin{algorithm}
\caption{Bitmap Version of \texttt{spMV}}
\begin{algorithmic}[1]
    \State \textbf{procedure} \texttt{spMV}
    \State \textbf{for} \textit{i} = 0; \textit{i} < \textit{n}; \textit{i} = \textit{i} + 1 \textbf{do}
    \State \textit{s} ← 0
    \State \textbf{for} \textit{j} = 0; \textit{j} < \textit{n}; \textit{j} = \textit{j} + \textit{c} \textbf{do}
    \State \textit{bits} ← \texttt{bitmap}[\textit{w} + \textit{j}]
    \State \textbf{for} \textit{k} = 0; \textit{k} < \textit{c}; \textit{k} = \textit{k} + 1 \textbf{do}
    \State \textbf{if} \textit{bits} & 0x1 \textbf{then}
    \State \textit{s} ← \textit{s} + \texttt{vals}[\textit{next}] \times \texttt{v}[\textit{j} + \textit{k}]
    \State \textit{bits} = \textit{bits} >> 1
    \State \textit{next} = \textit{next} + 1
    \State \textit{y}[\textit{i}] ← \textit{s}
\end{algorithmic}
\end{algorithm}

Array \texttt{bitmap}[.] holds the metadata of 0s and 1s, while array \texttt{vals}[.] holds the non-zero values of the original matrix. \texttt{v}[.] is the dense vector. Each iteration of the loop on loop index \textit{j}, the code fetches the next chunk of the bitmap (shown as of size \textit{c}, where \textit{c} could be 32-bit) and goes over each bit examining if it is a 1. If it is a 1, then the next value from \texttt{vals}[.] is multiplied with the corresponding element from \texttt{v}[.] and accumulated.

There are several performance overheads with this software-only approach. One, the metadata cost is high: each innermost loop iteration includes a check on the bit value before the actual multiply-accumulate can be performed. Two, unlike the traditional uncompressed matrix-vector multiplication algorithm, this sparse version has three nested loops thereby incurring additional loop control overheads. Three, the code is hard to parallelize/vectorize. As we will quantitatively discuss in Section 2, the \texttt{Bitmap} compression technique is not efficient for lower sparsity matrices. Similarly, almost all other compression techniques incur metadata overheads that often constitute a large fraction of processing cycles especially when the sparse matrices do not exhibit high sparsity. In fact, low to moderate sparsity (20% - 70%) is dominant in DNN workloads unlike the traditional scientific domain where sparsity in matrices is very high [6], [38]. Thus, while the sparsity needs to be leveraged for storage and energy gains, the performance drawback needs to be addressed.

In this work, we seek answers to the following: (i) at what sparsity levels is it worth eliminating compressed representation of matrices and use dense (uncompressed) representation of data that include both zeros and non-zero values, and (ii) even if we use compressed data representation, will it be useful to expand the matrices internally to eliminate metadata computations? In this context we propose \textit{ExPress} – a hardware accelerator. Denoted \textit{ExPress}, the accelerator’s goal is to simultaneously improve storage, energy and performance of low-sparsity matrix codes by expanding compressed matrices. Expanded data is presented to the CPU via memory buffers. In addition, \textit{ExPress} supplies mask bits to let the CPU skip wasteful computations. Thus, \textit{ExPress} simultaneously enables efficient storage as well as efficient computations by removing the metadata (or index) processing burden from software codes. We incorporate \textit{ExPress} support within the RISCV RV32IMCV ISA. We make the following contributions:

- Design and evaluate a novel memory-side accelerator that works in unison with matrix-based codes running on CPU cores. The accelerator improves performance of embedded single-threaded cores by eliminating sparse matrix metadata overheads and improving compute–memory overlap.
- Leverage the open RISCV RV32 32-bit core as our baseline to design the operation of the accelerator.
- Across a range of sparse matrices drawn from DNN workloads, and synthetic benchmarks, we demonstrate that \textit{ExPress} improves performance of \texttt{spMV} by 43%, 33% and 11% on average over Bitmap, Run-Length (RL) and CSR software codes respectively in embedded systems.
- \textit{ExPress} simultaneously achieves 15%, and 10% energy reduction over Bitmap and RL formats respectively by eliminating metadata processing cycles from processing elements.

2 BACKGROUND AND MOTIVATION

Intelligent real-time sensing applications such as keyword spotting [49] and visual wake word recognition [8] require real-time machine-learning based inference engines to execute on low-power sensors that are limited by power, storage and compute capabilities. On the low end of the compute spectrum, these microcontroller-based devices (MCUs) comprise simple in-order cores (such as a core from ARM Cortex-M series or RISCV RV32) integrated with a small on-chip SRAM (100Ks – a few MBs), clocked at no more than a few hundred million cycles per second. Thus, achieving intelligence at the edge requires highly optimized implementations of various types of ML inference algorithms.

Both Convolutional neural nets (CNN) and Recurrent neural nets (RNN) employ matrix-based algorithms for applications such as object detection [14], image captioning [28], speech recognition [16], and natural language processing [33]. With feature sizes and output classes in the order of a few thousand elements, the storage capacity required often exceeds available SRAM. For example, the final fully connected layer of VGG16 [45] employs a 4K x 1K weight matrix requiring a few MBs of storage. Thus, techniques such as network pruning and quantization have been used to reduce the memory footprint by eliminating some connections (weights set to 0) or by avoiding storing 0 values in intermediate features [19]. Unlike scientific sparse matrices, CNNs contain far less sparsity. Figure 2 plots the average sparsity of the output in each convolution layer of quantized implementations of VGG16, VGG19 and Cifar10. While sparsity improves with layer depth, earlier layers have only a moderate level of sparsity to exploit – more than half the layers have less than 60% sparsity.
Several sparse matrix formats have been proposed to achieve lossless compressed storage of sparse feature maps and weights [7, 18, 27, 32, 39, 40, 45]. While all of these formats achieve storage efficiency, they incur computational overhead to extract non-zero values using the matrix metadata, and sparse matrix algorithms are harder to vectorize/parallelize. Given the low to moderate sparsity seen in several convolution layers (see Figure 2), these compression overheads may annul the storage and computation savings expected with sparse data. Thus a wholistic analysis of the benefits of compression is needed, accounting not just for storage, but also for performance as well as energy. Below, we examine each of these for three representative formats – CSR, bitmap and run-length encoding, to motivate our ExPress design. Our goal is not to provide an exhaustive comparison of these formats but rather to provide motivation for jointly optimizing storage, performance and energy of sparse computations.

Storage Efficiency: Table 1 lists the storage requirements for various formats given an input $n \times n$ matrix with sparsity $s$ (sparsity refers to the fraction of data that are zeros – higher sparsity means more zeroes) and each non-zero element occupying $e$ bytes. The sizes shown assume bit-exact allocations – byte alignment or padding incur additional storage. Compressed Sparse Rows (CSR [2]) stores non-zero values of the sparse matrix in a row-major order. All the non-zero columns are stored in a CSR_Cols array ($sn^2 \log_2(n)$ bytes) and corresponding values in a CSR_Vals array ($sn^2 e$ bytes). A CSR_Rows array contains indices into the CSR_Cols array marking the start and end of the non-zero data for each row ($s n^2 e + sn^2 \log_2(n)$ bytes). In the Bitmap representation (see [40]), a bit map of the matrix is constructed where each bit corresponds to a matrix location and the bit value denotes if the location has a non-zero or zero value. An array of values similar to CSR_Vals is used to store the non-zero values. In the Run-Length representation (see [39]), a run of adjacent non-zero values is recorded by a pair ($\text{start\_column}, \text{num\_non\_zeroes}$). The average length of a run of non-zeroes is denoted by $l$. All such pairs are stored in an RL_Cols array ($sn^2 \log_2(n)$ bytes). Non-zero values are stored in an RL_Vals array (similar to CSR_Vals). An RL_Rows array contains the number of runs in each row ($n \log_2(n)$ bytes).

![Figure 2: Sparsity by Convolution Layer](image)

```latex
\begin{table}
\centering
\begin{tabular}{|c|c|c|}
\hline
Format & Size (Bytes) & Description \\
\hline
Uncompressed & $n^2 e$ & No metadata needed. Storage size does not depend on sparsity. \\
\hline
CSR & $(n + 1) \frac{\log_2(n^2)}{8} + \frac{sn^2 e + sn^2 \log_2(n)}{8}$ & $(n + 1) \frac{\log_2(n^2)}{8}$ is storage for CSR Row indices. The $\log_2(n^2)$ is the worst-case number of bits required to describe each index. $sn^2 e$ is storage for non-zero values. $sn^2 \log_2(n)$ is storage for CSR column indices of non-zero values. \\
\hline
Bitmap & $\frac{n^2}{8} + sn^2 e$ & $\frac{n^2}{8}$ is storage for the bitmap. $sn^2 e$ is storage for non-zero values. \\
\hline
Run-length & $\approx \frac{n \log_2(n)}{8} + \frac{sn^2 \log_2(n)}{8} + sn^2 e$ & The $\frac{n \log_2(n)}{8}$ is storage for number of RLs in each row. $\frac{sn^2 \log_2(n)}{8}$ is storage for (start, num) pairs assuming an average run length $l$. \\
\hline
\end{tabular}
\end{table}
```
While compressed formats lower the memory access energy, some energy of matrix-based computations, and (iii) processor pipeline energy is very low energy & power goals while implementing machine-learning algorithms. At the same time, ExPress processing metadata. This leads to both performance improvement and energy saving. At the same time, ExPress supplies mask bits to the CPU so that expansion does not result in unnecessary computations. Thus even though ExPress expands the compressed matrix, the energy incurred by computations (by computations, we mostly refer to multiplication operations that are prevalent in DNN workloads) is limited to useful non-zero data. Thus ExPress provides a memory-side substrate to orchestrate sparse computations that are simultaneously performance, energy and storage-efficient.

Figure 3 shows the system organization of a typical low-power MCU (diagram on the left) as well as a high-performance processor with ExPress(diagram on the right). The MCU is provisioned with a small amount of on-chip SRAM backed by non-volatile storage (typically flash memory). In the MCU, ExPress is integrated such that it can access the SRAM. We expect an on-chip interconnect through which ExPress can access the SRAM for (prefetching matrix contents). Memory read/write requests issued by ExPress are routed by the interconnect to the correct SRAM banks as per the chip-level memory map.

The high performance processor includes a cache hierarchy followed by off-chip memory (typically DRAM). ExPress is integrated into the L1D cache so that it can leverage the TLB to convert software-programmed virtual to physical address translation and issue memory requests for matrix contents.

ExPress is internally organized into a front-end (FE) and a back-end (BE). The FE is responsible for CPU-side interactions: handling configuration writes from the CPU and supplying data to the CPU in response to buffer load requests. By design, the front-end is unaware of the sparse format used. The BE is aware of the sparse format and uses the metadata to fetch values and provide their indices to the front-end. It issues loads of matrix data and metadata from the memory system to enable the FE assemble data buffers in a timely fashion. The FE and the BE operate in a decoupled manner synchronized by a control unit that starts or throttles the BE based on availability of space in the buffers. This separation between a sparsity-unaware front-end and sparsity-aware back-end enables the front-end portion of the design to be reused while only the back-end needs to be updated to support newer sparse formats.

### 3 EXPRESS

#### 3.1 Overview

ExPress is motivated by this storage–computation–energy trade-off at low sparsity. ExPress simultaneously exploits the storage reduction of sparse formats and the computational simplicity of the uncompressed format to achieve overall energy reduction. It does so by expanding the compressed data and supplying expanded data to the CPU. Expansion enables computational kernel software to become compression format-agnostic and the CPU does not incur cycles processing metadata. This leads to both performance improvement and energy saving. At the same time, ExPress supplies mask bits to

---

1Such on-chip interconnects are the norm in MCUs to allow DMA (Direct Memory Access) accesses to SRAM and we do not consider this as a new requirement for ExPress integration.
Figure 5: Matrix-Vector Multiplication: Energy Breakdown

- $M_{\text{Cols\_Base}}$: Base address of metadata array that provides information about non-zero locations in each row.
- $M_{\text{Values\_Base}}$: Base address of the array holding non-zero values of the matrix.
- $\text{Element\_Size}$: Size of each data value in the values array (to distinguish 8-bit/16-bit/32-bit/etc data types)
- $\text{Start/Stop}$: This bit is set to start or stop the hardware operation by the CPU.

Figure 7 provides an example of how the configuration registers are set up for CSR, Bitmap and Run-Length formats using the $3 \times 3$ matrix $M$ shown at the top of the figure. For the CSR format, the $M_{\text{Rows\_Base}}$ register contains the address of the $\text{CSR\_Rows}$ array; $M_{\text{Cols\_Base}}$ the address of the $\text{CSR\_Cols}$ array. With Bitmap representation, the $M_{\text{Rows\_Base}}$ register contains the address of an array that contains bit offsets at which each row’s bitvector is stored. $M_{\text{Cols\_Base}}$ contains the address of the bitmap of 1s and 0s to mark non-zero/zero locations in the matrix. In Run-Length, $M_{\text{Rows\_Base}}$ points to an array that contains the number of run-lengths in each row. $M_{\text{Cols\_Base}}$ points to an array describes each run-length as a pair: (number of non-zeros in the run, starting column index of the run). In all formats, $M_{\text{Values\_Base}}$ contains the address of the non-zero values array.

$\text{ExPress}$ provides hardware support for expansion of sparse matrices. It reads sparse matrix metadata and constructs buffers that are filled with either the non-zero values from the sparse matrix or zeroes. For each column of each row, $\text{ExPress}$ determines if the corresponding value is non-zero (value present in $M_{\text{Values\_array}}$) or zero using the specified matrix metadata. The CPU simply loads values from these buffers and multiply-accumulates them into the output vector. These buffer loads are performed via the normal load-store interface of the CPU using addresses assigned to these buffers. In our design, we assume a vector-wide load-store interface for high performance applications but $\text{ExPress}$ design can work with scalar load-store interfaces also. The software uses a fixed load address to load from. Whenever the CPU performs a load, the $\text{FE}$ updates its buffer state to determine when the buffer has been completely drained by the CPU. If the $\text{FE}$ is designed with multiple buffers, then upon the CPU draining one buffer, the $\text{FE}$ switches to the next ready buffer. In the single-buffer design, as buffer entries are read, these slots are filled with the next chunk of data. In this sense, the $\text{FE}$ offers a streaming FIFO interface to the CPU. If the CPU performs a load when the buffer is not ready, then the $\text{FE}$ stalls the load. In both the MCU and High-performance processor integration, it is assumed that the CPU can be stalled due to a long latency memory access.

Eliminating Wasteful Multiplications: While $\text{ExPress}$ eliminates the software overhead of processing the metadata, it increases the number of multiplications performed by the CPU since it supplies both non-zeros as well as zeroes. This is wasteful especially from an energy perspective in embedded systems. In order to mitigate this, the $\text{FE}$ supplies a hint in the form of a bit-vector to the core alongside the buffer data indicating which buffer elements are non-zero or zero. This is depicted in Figure 8. Using the bitmap representation as example, the figure shows the $\text{FE}$ expanding a chunk of data by inserting two zeroes and supplying mask bits to the core in addition to the actual data via the buffer.

In our design, this mask-vector is supplied as a side-band signal to the load-store interface of the CPU. The CPU uses the mask to

---

Footnote: In fact, our design works even better with scalar loads as there is less pressure on the memory system to return a large number of values per loop iteration.
enable/disable operations on individual elements of the loaded data. While this causes an idle slot in case an operation is skipped, as our results demonstrate, ExPress simultaneously improves performance and lowers energy by a combination of compressed storage, and expanded, zero-skipped computations.

**FE Design:** The FE is implemented with \( N \) vector-sized buffers where \( N \) is a design-time parameter. \( N \geq 2 \) permits ExPress to prefetch data and fill buffers ahead of time. \( N = 2 \) provides double-buffer arrangement. The FE and BE work the memory pipeline managed by a control unit. Figure 9 describes the design of the ExPress front-end.

The first stage of the pipeline reads the next non-zero column index (supplied by the back-end, discussed in Section 3.3). Next, it calculates the gap between this index and previous index. This requires a comparator. If there is no gap, then the value of the matrix element at this index is read (which is supplied by the back-end). If a gap is found, then a zero value is inserted into the pipeline. In the final fill-buffer stage, the value (either the matrix value from memory or zero) is written to the next free slot in the output buffer. The control unit maintains the current state of the FE to issue pipeline control signals. It also tracks the current read and write positions in the buffers so that CPU read requests are serviced in correct order. The control unit also tracks buffer empty/full conditions so as to stall CPU load requests (when no ready buffer is available), skip issuing new memory read requests when all buffers are full, etc. This internal state requires a single 32-bit register inside the control unit (active read buffer id, active write buffer id, next read slot in read buffer, next write slot in write buffer, empty read buffers flag, full write buffers flag).

When the pipeline is working at its maximum efficiency, a new value is filled every cycle. This throughput is sufficient to feed a CPU operating at the same frequency even if the CPU uses SIMD execution with 8 element-wide vectors. We limit our analysis to a maximum of 8 element vectors as wider data paths are area and power-expensive and rarely implemented in low-power embedded systems.

### 3.3 ExPress Back-End

ExPress Back-End (BE) fetches metadata and data for the front-end by interpreting the sparsity format that has been programmed into the Sparse_Format memory-mapped register. Figure 10 describes the pipeline design of the ExPress back-end. Rather than describing the BE design and operation for each format separately, we have described a generic pipeline-based organization with some pipeline stages providing format-specific functionality.

The first stage calculates the next metadata address to read from. This address calculation depends on the format used and the current read position. For example, in the Bitmap format, the next address is simply the next 32-bit word in the array pointed to by M_COLS_Base register. With CSR and Run-Length formats, the next
address depends on whether the current read position is the end of a row: if it is, then the next address is the next entry in the array pointed to by M_Rows_Base; if the read is in the middle of a row, then the next address is the next entry in the array pointed to by M_Cols_Base. The calculated address is used to issue a memory read request in the second stage. In the third stage, the next data value is read from memory using an offset from the address held in M_Vals_Base. In all formats, the address generation to access the values array is trivial: the $i^{th}$ access is simply to the address $M_{\text{Vals Base}} + i \times \text{Element Size}$. The control unit maintains necessary state – the current read positions into the rows, columns and values arrays. In the fourth stage, the next non-zero column index is calculated. The calculation performed is a function of the format used: in CSR, the column index is explicitly read from memory and no calculation is needed; in Run-Length, the column index is an increment of one from the current (previous) column index in the current run; in Bitmap, the 32-bit column metadata bitvector is scanned for the next ‘1’ bit and its bit position is converted to column index. In the final stage, the calculated column index value and the corresponding data value are supplied to the front-end by skipping these values to registers. The control unit generates signals to control the pipeline movement and to stall the pipeline if memory responses are delayed or if the next non-zero column index could not yet be computed (this can occur in the Bitmap format - where a string of 0s-only words may be encountered).

The BE works with the underlying memory system to issue read requests and to collect read data. In the MCU integration, the BE issues requests to the on-chip RAM via an on-chip interconnect. In the high-performance processor integration, the BE issues requests to the L1D cache. If the request is an L1D miss, then the usual cache miss processing is carried out to fetch the contents.

Depending on underlying cache hierarchy and memory organization, the BE may reorder memory requests to improve spatial locality. In particular, on DRAM-based systems, the BE may employ techniques such as prefetching data from open row-buffers to reduce memory access latency and improve the overall efficiency of the memory system.

3.4 Other Considerations

Multi-core Support: ExPress is designed to work on a per-core basis. In multi-core implementations of matrix algorithms, per-core ExPress instances can be configured to accelerate respective cores. It is straightforward to extend ExPress to support matrix sub-blocking so that each per-core instance manages supplying data accessed by one core. In a multi-core implementation, sub-block dimensions have to be programmed into each ExPress instance (rather than the entire matrix dimensions).

Other Compression Formats: Like most prior works, ExPress focuses on several commonly used formats: CSR, Bitmaps and Run-Length. Support for other compression formats can be integrated into ExPress with relatively minor changes only to the step that determines the location of the next non-zero element in a row, column or sub-block. The rest of the architecture including the streaming buffers and memory-side back-end remain the same.

Saving and Restoring ExPress State: The ExPress state may be treated as part of the process state of the process that is currently using ExPress. If the process is context-switched out, then the state of ExPress could be saved along with the CPU state (PC, stack, etc). ExPress state comprises the filled-but-unread buffers and the metadata bookkeeping registers. The restoration of process state is similar: the unread buffers and state of ExPress metadata are restored. Once this is completed, the start-bit may be set to resume operation.

Like the handling of floating-point state, an optimization is to let the OS determine if ExPress is being used by the current process. If it is not being used, then reading and saving ExPress state can be skipped, thereby saving context-save and restore latency for such processes.

Exceptions: ExPress is expected to be correctly configured by the application with correct base addresses of data and metadata arrays. Incorrect configuration can result in illegal memory accesses. Detection and management of such exceptions is outside the scope of ExPress and should be handled by the application core.

Cache Coherence and Buffered Data: Buffered data is not expected to go stale. While a thread is consuming matrix data for spMV or spMspM, it is not expected that the same or another thread of the process modifies these values. If such a scenario did arise, then the ExPress BE could be augmented to participate in cache coherence. It may also be observed that ExPress does not modify any data and therefore need not issue ownership requests.

Output Compression: At present, ExPress is designed for constructing input dense buffers for the CPU to load. The creation of compressed outputs is performed in software. It is possible to extend ExPress such that it can take a dense buffer from the CPU (via a vector-store to a ExPress buffer address) and suitably re-format and write the compressed output to memory.

Support for High-Dimensional Tensors: While this work explores ExPress in the context of the spMV algorithm, it is straightforward to extend the concept of expansion to cover higher dimensional data such as tensors. If the neural network algorithm executing on the processor is operating in chunks of 3D volumes of feature maps, then ExPress buffers should be constructed to supply these chunks.
3.5 Area Estimate

The area of ExPress is a sum of the logic gates of the control unit and storage required by the FE and the BE.

FE Area: The FE comprises CPU-side buffers (N × 32B), memory-mapped registers (8 × 4B), internal state registers (4B), pipeline registers (8B between successive stages) and column-index & value storage between the BE and FE (2 × 4B). With a single buffer (N = 1) of size 32B, the total storage is less than 100B. Coupled with a comparator in the control unit for gap calculation, we obtained an area estimate of 0.03 mm² using CACTI [30] with 32nm technology.

BE Area: The BE comprises control unit state for read pointers (4B each for row, column and values arrays), pipeline registers (8B between successive stages) and logic for address generation (adder), end-of-row comparison (comparator) and column index computation (priority encoder & shifter). The total storage needed is less than 50B. The area estimate for the BE is 0.03 mm².

The total area for ExPress logic and storage is an estimated 0.05 mm². In comparison to a RISCV 32-bit 3-stage in-order core with a vector unit with 32 32B vector registers without floating/double-precision support, this is less than 9% area overhead. In more sophisticated pipelined cores with floating point support, the area overhead of ExPress is insignificant.

4 PROGRAMMING MODEL

In order to leverage ExPress, it has to be configured with matrix metadata. This is done by programming memory-mapped registers. These registers provide the following configuration details (to be supplied by software):

- M_Num_Rows: Number of rows of sparse matrix M.
- M_Num_Cols: Number of columns of sparse matrix M.
- Sparse_Format: Format in which the sparse matrix is stored (one of CSR, Bitmap, Run-Length in our evaluation)
- M_Rows_Base: Base address of metadata array that provides aggregate information about each row.
- M_Cols_Base: Base address of metadata array that provides information about non-zero locations in each row.
- M_Values_Base: Base address of the array holding non-zero values of the matrix.
- Element sizes of M_Rows, M_Cols and M_Values Arrays: Quantization and pruning of ML networks leads to different reductions – such as 16-bit values or 8-bit column indices (in small-sized matrices or in small run-length encodings). Thus ExPress is programmed with the sizes (in bytes) of the elements of the various arrays that it accesses. This allows ExPress to seamlessly support various configurations of supported sparse formats.
- Start: This bit is set last to trigger the start of the hardware operation.

Software is expected to set up these configuration registers with the Start bit being the last one to be set.

4.1 Accessing Data Buffers In Computational Kernels

The ExPress data buffer is memory-mapped to a fixed address. Note that even if ExPress is implemented with multiple physical buffers for "double buffering" operation, the CPU software accesses the current buffer via the same address. In this sense, ExPress offers a streaming FIFO (First-In First-Out) interface to the CPU: the CPU software need not keep track of which buffer to read from. The software always issues a load (or vector-load) from a fixed address. ExPress routes this load request to the correct buffer and returns load data.

Putting it all together, Figure 11a compares and contrasts the traditional software-based spMV with ExPress-based implementation shown in Figure 11b. The traditional software implementation uses the Run-length format.

For simplicity of illustration, both codes are shown as scalar implementations while implementations may use RISCV vectors to accelerate the kernel. It may also be noted that vectorizing the software version is non-trivial/has lower gains due to the dependence on processing the run-length metadata before the actual multiplications can be vectorized. The code on the left is the traditional spMV where software processes run-length metadata comprising number of runs in each row (M_Rows), and description of each run (M_Cols). The code outline on the right is the ExPress-accelerated version of spMV. Relevant changes are highlighted as blue text. "The volatile pointer variable BUFFER is initialized to the address of the ExPress buffer^3. Next, before executing the main kernel, ExPress is initialized and started. These initialization functions (not shown for brevity) simply perform a series of writes to ExPress configuration registers. The main kernel of the ExPress-version looks very similar to the simple uncompressed matrix-vector algorithm. The only change is the way that values from M are accessed. In the ExPress-version, these values are accumulated into buffers by ExPress and software reads them from the buffer using the BUFFER pointer^4.

It may be observed that metadata overheads have been entirely eliminated by ExPress leading to simpler more efficient inner loops. While the above description used the matrix-vector kernel, the same programming model applies to other sparse-matrix based kernels such as convolutions.

5 EXPERIMENTAL EVALUATION

We evaluate ExPress on both low-power MCUs as well as high-performance processors using DNN, and synthetic workloads executed on a 32-bit RISCV ISA using a heavily modified Spike [15] simulator.

5.1 System Configurations

We evaluate ExPress with two different types embedded systems configurations. Tables 2 and 3 describe the system configurations of the low-power micro-controller and high-performance processor respectively. Both configurations use the 32-bit RISCV [11] base architecture along with vector (V), compressed (C), atomic (A), multiply (M), floating (F) and double precision (D) extensions.

The micro-controller (MCU) uses an in-order three-stage pipeline implementation. In particular, loads that do not complete in a single cycle stall the pipeline. The vector unit is not pipelined. The memory comprises on-chip SRAM. All the code, global data, stack and dynamically allocated data reside on SRAM. The high-performance

---

3In our experiments, we mapped the ExPress buffer to address 0xC000_1000.
4The volatile attribute ensures that each read goes to memory.
5.2 Workloads

We use several matrices corresponding to the fully connected (FC) layers of trained DNNs, and synthetically generated matrices in order to evaluate ExPress. The fully connected layer of DNNs performs matrix-vector multiplication before the final classification is performed. We leveraged the quantized weights matrix of this layer from a variety of networks: MobileNet [22], MobileNetV2 [44], DenseNet [23], ResNet [20], ResNetV2 [21], and VGG16, & VGG19 [45]. Table 4 lists attributes of interest for these matrices. As these networks are trained to classify input images into one of a set of 1000 pre-trained classes, the number of columns for each network's FC layer is 1000. As we will see in Section 6, a combination of average sparsity and average run-length greatly affect the performance of sparse-format-based software implementations.

In order to analyze the performance of our accelerator more carefully, we generated 28 synthetic matrices comprising 4 different sizes (64 by 64, 256 by 256, 1024 by 1024 and 4096 by 4096) and 5 different sparsity levels (10% through 70% in steps of 10%). We limit

<table>
<thead>
<tr>
<th>Processor</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>RISC-V with IMACFDV Extensions</td>
</tr>
<tr>
<td></td>
<td>Frequency = 2 GHz</td>
</tr>
<tr>
<td></td>
<td>In-order 3-stage</td>
</tr>
<tr>
<td></td>
<td>Vector width (VL) = 8 Elements</td>
</tr>
<tr>
<td></td>
<td>Element Size (SEW) = 32 bit</td>
</tr>
<tr>
<td></td>
<td>Vector Arithmetic Latency = 4 cycles</td>
</tr>
<tr>
<td>ExPress</td>
<td>Buffer size = 32B</td>
</tr>
<tr>
<td>MEM</td>
<td>SRAM, 1-cycle access time</td>
</tr>
<tr>
<td>Energy (pj)</td>
<td>Instruction Fetch: 5</td>
</tr>
<tr>
<td></td>
<td>16-bit Multiplication: 5</td>
</tr>
<tr>
<td></td>
<td>SRAM Access: 30</td>
</tr>
</tbody>
</table>

Table 2: System Parameters: Low-power Microcontroller

<table>
<thead>
<tr>
<th>Processor</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>RISC-V32 with IMACFDV Extensions</td>
</tr>
<tr>
<td></td>
<td>Frequency = 2 GHz</td>
</tr>
<tr>
<td></td>
<td>In-order 3-stage</td>
</tr>
<tr>
<td></td>
<td>Vector width (VL) = 8 Elements</td>
</tr>
<tr>
<td></td>
<td>Element Size (SEW) = 32 bit</td>
</tr>
<tr>
<td></td>
<td>Vector Arithmetic Latency = 4 cycles</td>
</tr>
<tr>
<td>ExPress</td>
<td>Buffer size = 32B</td>
</tr>
<tr>
<td>MEM</td>
<td>SRAM, 1-cycle access time</td>
</tr>
<tr>
<td>Energy (pj)</td>
<td>Instruction Fetch: 5</td>
</tr>
<tr>
<td></td>
<td>16-bit Multiplication: 5</td>
</tr>
<tr>
<td></td>
<td>SRAM Access: 30</td>
</tr>
</tbody>
</table>

Table 3: System Parameters: High Performance Processor

<table>
<thead>
<tr>
<th>DNN</th>
<th>Size</th>
<th>Sparsity (%)</th>
<th>Run-Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>DenseNet</td>
<td>1024 × 1000</td>
<td>49</td>
<td>11.2</td>
</tr>
<tr>
<td>MobileNet</td>
<td>1280 × 1000</td>
<td>11</td>
<td>8.9</td>
</tr>
<tr>
<td>MobileNet</td>
<td>1024 × 1000</td>
<td>30</td>
<td>3.3</td>
</tr>
<tr>
<td>ResNet</td>
<td>2048 × 1000</td>
<td>53</td>
<td>1.9</td>
</tr>
<tr>
<td>ResNetV2</td>
<td>2048 × 1000</td>
<td>34</td>
<td>3.9</td>
</tr>
<tr>
<td>VGG16</td>
<td>4096 × 1000</td>
<td>12</td>
<td>7.8</td>
</tr>
<tr>
<td>VGG19</td>
<td>4096 × 1000</td>
<td>12</td>
<td>7.9</td>
</tr>
</tbody>
</table>

Table 4: DNN Workload Sparsity Characteristics
our attention to these configurations as they correspond to DNN weight matrices in terms of both sizes and sparsities.

5.3 Simulation Details
We used a heavily modified version of Spike simulator [15] for our work. Spike models the 32-bit RISCV architecture along with several extensions that we required for our work, including vectors, compressed, atomic, multiply, floating and double-precision extensions. Further, Spike models a simple 3-stage in-order pipeline that closely resembles the implementation of most embedded microcontrollers. For simulating the high-performance processor, we incorporated several modifications to the baseline simulator including a detailed DRAM memory model and processor wait cycles. We also incorporated configurable instruction latency for vector instructions. ExPress is implemented as a detailed timing C model integrated at the load-store interface and occupying the address range (0x00000000, 0x00000000). It supports expanding from CSR, Bitmap and RL compression formats in a timing-accurate manner – each clock cycle, it uses a state machine to take suitable action (respond to CPU read request, store memory read response into buffer, issue next memory read request, etc). In addition, we extended Spike to support additional performance counters such as processor wait cycles due to memory, and memory access statistics.

6 RESULTS
We first present the performance results of ExPress on the embedded MCU configuration followed by its results on embedded high-performance processors. In terms of notation, ExPress-Bitmap denotes ExPress working with Bitmap format, and similarly ExPress-RL and ExPress-CSR.

6.1 ExPress on MCU
Figure 12 plots the performance improvement achieved by ExPress over respective software sparse codes on DNN fully connected layers. On average, ExPress improves performance by 43%, 11% and 33% over Bitmap, CSR and RL-based software codes. ExPress Bitmap consistently outperforms Software-Bitmap which is interesting since Bitmap is a commonly used compression format in several accelerator-based approaches (such [27]). Programmable software-only DNN approaches should consider this while choosing the appropriate compression format. ExPress-RL almost always outperforms RL except in denseNet. Referring to Table 4, denseNet has a high average run-length (11.2) which reduces the software overheads of processing run-length metadata. This result also reveals that run-length coding has significant performance variation that is dependent on the average run-length of the input. On resNet which has the lowest average (1.9), ExPress-RL outperforms ExPress-Bitmap. ExPress-CSR exhibits the least gain among the three formats. This is explained by Figure 4 which shows that CSR has very low metadata overhead. Thus, when the sparsity is reasonably high, software-CSRs performs better than ExPress-CSR since the software version avoids redundant processing altogether. denseNet and resNet have higher sparsity (49% and 53% respectively) and are thus able to take advantage of the CSR formatted metadata more effectively. However, CSR incurs higher metadata storage overheads which become more pronounced with higher quantization (such as 8-bit). For example, at 50% sparsity in a 1024 x 1024 matrix comprising 8–bit data, CSR requires more storage than a fully uncompressed matrix. Thus, depending on system constraints, one or the other format may be better suited. ExPress performs consistently – it is able to deliver performance improvements in 18 out of 21 configurations evaluated.

Energy Savings: While ExPress uses the same underlying storage format as the respective sparse format and thus incurs the same memory energy, it saves energy by removing instruction execution overheads. Unlike software codes that incur cycles to decode/decompress sparse data, with ExPress, the CPU executes fewer total instructions thereby saving energy. Figure 13 plots the energy saved by ExPress. Tracking performance improvement results presented above, energy savings are the highest in ExPress-Bitmap owing to the high metadata overheads of the Bitmap format (average saving of 15%). Similarly, ExPress-CSR achieves only a modest average improvement as Software-CSRs is quite metadata-efficient.

6.2 ExPress on High-Performance Processor
Figure 14 plots the performance improvement achieved by ExPress over traditional software sparse codes on HP cores running DNN spMV operations. On average, ExPress performs 43%, 37% and 6% better than traditional Bitmap, Run-length and CSR formats respectively. In general, results on HP CPUs follow similar trends as on MCU CPUs. Software-CSRs outperforms ExPress when sparsity is higher while ExPress performs better than the other two formats consistently.

In our design, the ExPress hardware issues memory accesses in parallel with CPU computations, thus hiding memory access latencies. However, since the CPU now performs uncompressed matrix computations, it will be performing computations over all values (even though zero values are skipped, a wasted cycle is still incurred). The sparsity determines if the memory savings outweigh computational overhead or not. In addition to sparsity, the size of the matrix also determines the trade-off. At a given sparsity, larger matrices involve more redundant computations than smaller matrices (see Figure 15 and the discussion in section 6.3).

Thus, it is necessary to understand the sparsity of data in a given application and the size of the data to determine if a design like our ExPress is beneficial or not.

6.3 Analysis on Synthetic Matrices
In this section, we evaluate ExPress on several synthetic matrices of different sizes and sparsities to understand the limits of ExPress. Since ExPress performs computations in uncompressed format, as sparsity increases, wasted computational cycles can outweigh the gains achieved by metadata overhead elimination. Figure 15 shows the performance improvement of ExPress as compared to respective software sparse formats for two matrix sizes (i.e. 64x64 and 2048x2048) at different sparsity levels. Unsurprisingly, it can be seen that the performance benefit of ExPress drops with increase in sparsity. As the matrix size increases, performance drops at a relatively faster rate with increase in sparsity because of the increase in wasted cycles. It may also be observed that ExPress gains at least 20% improvement over both Software-Bitmap and Software-RL even...
6.4 Compute-Memory Overlap

ExPress improves compute-memory overlap by issuing memory requests in parallel to CPU computations. By provisioning more than one buffer, it is possible to improve this overlap further and hide memory wait cycles. With \( N = 2 \) buffers, we noticed \( 1 - 2\% \) performance improvement over the single-buffer configuration. In our experiments, as the core performs scalar operations, a single buffer is largely sufficient to meet the load bandwidth of the CPU. However, we expect higher gains when ExPress is deployed in multi-core or vectorized systems where the processing elements can consume several data elements per cycle and thus multiple buffers can be beneficial.

ExPress offers another benefit: vectorization. By rendering uncompressed matrices, it becomes trivially easy to vectorize matrix kernels unlike kernels that are based on compressed formats. We implemented a vectorized version of ExPress-CSR and compared its performance improvement over vectorized-software-CSR. For

---

**Figure 12:** ExPress on MCU CPUs: DNN workloads

**Figure 13:** ExPress on MCU CPUs: Energy Saving
a 2048 × 2048 matrix with 30% sparsity, ExPress-CR improved performance by 56% (in contrast, scalar ExPress-CR improved performance by only 17% over scalar-software-CR). It can be seen in Figure 16 that the vectorized version of DNN workloads has similar performance improvements as scalar counterparts.

7 RELATED WORKS

Accelerating \( spMV \) 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 \([48]\), and an accelerator for very large \( spMV \) \([43]\). The work in \([43]\) 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 aim to solve the memory latency problem faced by embedded system-based matrix codes. Unlike works that
aim to move the entire computation to a dedicated accelerator, our

goal is simply to reduce the decompression bottleneck faced by

software codes running on traditional cores.

Interest in DNN based accelerators have seen a rise in recent

years. There are too many different hardware/software implemen-
tations to include here. Many are based on specialized accelerators

based on either dataflow or tensor/systolic arrays. Many of these

systems lack flexibility or reconfigurability. A recent paper [40]

focuses on support for flexible sparse matrix and vector multipli-
cations. Sparse data is represented as bit-vectors and dataflow like

Multiply-Accumulate units are configured based on the nonzero

values in data. Authors of [34] propose a programmable accelerator to

optimize the execution for new and emerging ML applications. The

accelerator (VTA) is viewed as a fetch-load-compute-store pipeline to

dispatch instructions to load (obtain input, weights and bias tensors from DRAM), compute (GEMM operations) or store (store results of compute in DRAM). Our interest is in the use of general

purpose RISC-like processing units with minimal extensions to the

ISA and hardware complexity.

There are several works that attempt to improve the performance

of sparse matrices for scientific applications. Authors of [5] pro-

posed a parallel sparse matrix algorithm based on SUMMA used in

BLAS library and parallelized the sparse matrix multiplication, while we used ExPress to expand low-sparsity matrices to remove

the metadata burden from CPU codes. Greathouse [17] proposed an

algorithm, CSR-Stream to compute sparse matrix-vector multipli-
cation for smaller rows and CSR-Adaptive algorithm to choose

CSR-Stream instead traditional CSR compared to expansion from

COO format and parallelizing dense matrix workloads. In [1], au-

thors proposed a parallel Sparse Matrix-Sparse vector (SpMSpV)

algorithm that stores the product of Sparse matrix-vector based on

the row indices and later accumulates it, all by using buckets.

There have been many studies on near-data processing Processing-

In-Memory logic. More recent works focused on migrating com-

putations to PIM. Some older reports proposed migrating memory

intensive operations closer to memory including memory allocation

and garbage collection functions (see for example [9, 42, 47]). 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].

There are several studies on data prefetching. In [13], prefetching

is based on calculating the stride from previous accesses and

prefetching is limited to tracking the stride for one data structure.

Streambuffers [26] prefetch sequential streams of cache lines even

if the fetched data is not utilized, while our work prefetches only

useful data elements and supplies them to processing elements.

Markov prefetching [25] supports correlation-based prefetching

by storing the history of missed address streams. Based on the

history of previous miss patterns, future misses are predicted and

prefetched. This method does not maintain any knowledge about

specific data structure strides or keep track of multiple structures

simultaneously. In [24], authors prefetch data based on distance

prefetching from slower memory into on-chip buffer in a heteroge-
nous memory architecture (consisting of faster 3D DRAMs and

slower non-volatile devices such as PCM). The distance is measured

in terms reuse distance. The authors propose to prefetch heavily

used pages from slower non-volatile memories into faster DRAM

based memories. This is in lieu of migrating pages completely into

faster memories.

In a different vein, there have been proposals on improving

compression of sparse matrices and proposed techniques include

CSR5 [31], hierarchical bit vectors [27], compression on top of

CSR [41], hierarchical coordinate format [29] and the structured

2:4 format [35]. Some proposed specialized hardware to compress

decompress data for use by CPU (assuming that the CPU uses

conventional spMV software, relying on CSR formats) [41]. Our

work could be leveraged on top of these other formats in order to

offer both storage benefits as well as compute efficiency.

8 CONCLUSIONS

Matrix-Vector multiplication is inherent in many scientific, graph

analytics, machine learning and deep Neural network applications.

In many cases, the matrices are sparse, although the sparsity lev-

eels (the fraction of data that are zeros) varies. There have been

many different ways of representing the sparse matrices to save the

storage needed for the data, including Compressed Sparse Rows

(CSR), Bitmap, Run-Length encoding and hierarchical representa-

tions. While these representations lead to storage savings, they can

lead to computational overheads since it is necessary to identify

the location of rows and columns of non-zero elements of the ma-

trix and match the corresponding vector elements needed for the

computations. This led us to investigate answers to two questions:

(1) is it better to represent matrices in dense format to improve

computational efficiency, even if this leads storage overheads, and

at what sparsity levels this approach is desirable, and (2) even if

matrices are represented using a sparse representation, is it better

to expand them internally to dense representations before computa-

tion proceeds, and at what sparsity levels is this approach desirable.

In response to the first question, we have shown that dense repre-

sentations are viable at certain sparsity levels and the sparsity

level depends on the size of the matrix. For example for 1024×1024

matrices dense representation outperforms CSR-based algorithms

for matrices with up to 40% sparsity. To answer the second question,

we designed a special hardware called ExPress that expands sparse

data into dense data so that the CPU performs dense Matrix-Vector

computations. We find that our approach outperforms CSR-based

algorithms for matrices with 40% sparsity or less, and outperforms

Bitmap based algorithms for matrices with 70% sparsity or less.

In addition to performance gains, ExPress also leads to energy savings for

matrices with low to moderate sparsities, as is the case with many Deep Neural Network workloads. Further, ExPress simplifies

the programming model enabling vectorization and unrolling opti-

mizations. As future work, we are considering hardware support for

supplying only non-zero index-aligned values of matrix rows and

vector instead of expanding sparse matrices at higher sparsities.

9 ACKNOWLEDGEMENTS

This research is supported in part by the Semiconductor Research

Corporation (SRC) under SRC AIHW Task 2943. The research is

also supported in part by NSF award #1828105.
for DNN training. (Feb 2020).


