HHT: Hardware Support For Accelerating Sparse Matrix Algorithms

ABSTRACT
Matrix algorithms such as matrix multiplication, matrix-vector product and convolutions have witnessed a resurgence with ever-expanding application of deep neural networks and other machine learning algorithms. In many cases, matrices are sparse, and leveraging sparsity of data enables efficiency of storage and computation by avoiding storing and processing zeroes. However, sparse representations incur metadata computational overheads – software first needs to find row/column locations of non-zero values before performing necessary computations. Such metadata accesses involve indirect memory accesses (of the form \(a[b[i]]\) where \(a[]\) and \(b[]\) are large arrays) and they are cache and prefetch-unfriendly, resulting in frequent load stalls.

We propose Hardware Helper Thread (HHT) – a memory-side accelerator for sparse data computations in embedded or edge computing devices. HHT works with embedded cores by relieving computational cores of metadata processing and management. HHT provides a simple streaming interface by supplying the cores with appropriately aligned non-zero values. This reduces dynamic instruction count and improves compute-memory overlap resulting in performance gains for sparse data computations. We only accelerate index accesses unlike other studies that accelerate the entire computational kernel. In this paper we will explore using both a dedicated hardware for HHT and a simple RISC-like programmable core. We show both performance gains and energy savings of HHT for sparse matrix-dense vector multiplication (SpMV), sparse vector-sparse vector multiplication (SpMSpV) and convolutions with sparse input feature data. With sparse matrix-vector multiplication algorithm, our ASIC-HHT shows 1.73 times speedup on average and Programmable-HHT shows 1.24 times speedup on average over vectorized software-only baselines running on low-power micro-controllers. The dedicated ASIC HHT was designed using System Verilog and implemented using 28nm Arm library in Synopsys Design Compiler-2020.09 and compared with PULP v1 like RISCV processor. It was observed that ASIC HHT requires less than 1% of a RISCV core area while saving 55% of energy consumed for SpMV.

Keywords: Sparse matrices, DNN, Hardware Accelerators, RISCV

1. INTRODUCTION
With the trend towards embedding intelligence into the edge, there is a growing need to architect 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 memory [37, 38, 51]. Achieving real-time inference capability in these devices requires optimizing both the storage and computations performed. Leveraging sparsity (zeroes) in the data and/or weights of deep neural nets (DNNs) has emerged as a viable technique to achieve these improvements [24, 39, 55].

Sparse matrix-vector multiplications (SpMV) are at the heart of machine learning, data analysis and scientific applications. Algorithms such as forward and back-propagation in deep neural nets [50] & graph neural nets [53], Markov clustering [5], high-dimensional similarity search [2], topological similarity search [26], clustering coefficients [4], betweenness centrality [10], multi-source breadth-first-search [23], label propagation [41], and solvers of discretized differential equations [47] employ matrix-vector multiplications involving sparse matrices. Sparsity (the percentage of zeroes in the matrix) can be exploited to improve performance, as well as reduce storage and energy requirements. We need to distinguish between sparse matrix and dense vector, (denoted SpMV) and sparse matrix and sparse vector (denoted as SpMSpV). SpMV algorithms only require the column indices of non-zero elements (in rows) of Matrix to find needed Vector elements. However, SpMSpV requires the alignment of non-zero elements of Matrix with non-zero elements of the Vector.

Various sparse matrix representations have been proposed and incorporated in scientific and machine learning codes. These include compressed sparse row (CSR [7]), block compressed CSR (BCSR [8]), compressed sparse column (CSC [9]), coordinate list (COO [15]), bit-vectors [39], and run-length encoding [39]. There are also some newer representations including hierarchical bit vectors [32] and compression on top of CSR [42]. Conceptually, compressed representations store only the 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 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. 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. If the memory itself (or a small processing unit placed close to memory) could perform this metadata access and provide only needed non-zero 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 memory-side hardware called **Hardware Helper Thread (HHT)**.

In this work we make the following contributions:

1. We use a memory-side accelerator, called Hardware Helper Thread (HHT), to assist primary computational core in locating non-zero values. We evaluate both a dedicated (or ASIC) implementation of HHT and a programmable HHT using a small RISCV core.

2. We evaluate HHT for both sparse Matrix - dense Vector (SpMV) and sparse Matrix - sparse Vector (SpMSpV) multiplications. We also show the use of HHT for implementing convolution codes.

3. Programmable cores can be used to handle both known and possible newer sparse data representations. However, ASIC HHT provides greater performance gains. The ASIC hardware HHT achieves a speedup of 1.8 times for SpMV for matrix sparsities ranging between 10% to 90%, up to 4 times speedup for SpMSpV and 2 times speedup for convolution algorithms. The performance gains achieved using programmable RISCV cores are, as one would expect, smaller.

4. We provide detailed design and implementation of our dedicated hardware HHT for handling CSR representations. We analyze the silicon area needed for our HHT is less than 1% of RISCV, estimate power consumed as 0.7930 mW compared to 115.4 mW of RISCV and energy savings (on average 55%) achieved using HHT augmenting a RISCV core executing SpMV computations when compared to a baseline RISCV core executing the same computations.

2. MOTIVATION FOR ACCELERATING INDEX COMPUTATIONS

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. 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 bit-vectors. In the CSR representation, a $cols$ array holds the column indices of the non-zero values in each row. 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. In the bit-vector representation, an array of bits is used, one bit for each matrix cell. A 1 denotes a non-zero value in that cell.

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.

![Figure 1: A 3x3 sparse matrix in CSR and Bit-Vector Formats](image)

**Algorithm 1 CSR Version of spMV**

```plaintext
1: procedure SpMV(M_rows, M_cols, M_vals, n, v)
2: $k \leftarrow 0$
3: for $i = 0; i < n; i = i + 1$
4: $n_{z} \leftarrow M_{rows}[i+1] - M_{rows}[i]$
5: $s \leftarrow 0$
6: for $j = 0; j < n_{z} ; j = j + 1$
7: $s \leftarrow + M_{vals}[k+j] \times v[M_{cols}[k+j]]$
8: $k \leftarrow k + n_{z}$
9: $y[i] \leftarrow s$
```

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. Processor vendors have offered vector gather instructions (see Intel [31], ARM [49] and RISCV [17]) for software codes to issue requests to the memory system to gather elements from an array using array indices supplied in a vector register. While these indexed vector loads help specify the gathering operation to the memory system, they do not provide the memory system enough of a look-ahead: the memory system cannot prefetch data for future requests as it has no visibility to the future array indices that will be requested. Given the random nature of the indices accessed, traditional prefetchers perform poorly. Even with perfect memory accesses, the indirect accesses increase the dynamic instruction count of an otherwise efficient nested loop. We label the accesses to the $cols[.]$ array and subsequent use of these values in fetching values of $v[.]$ as metadata overhead.

Our focus is on the use of 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, 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-vector multiplications for applications such as object detection [19], image captioning [33], speech recognition [21], and natural language processing [35]. In a typical CNN architecture, the last 1–2 layers are fully connected layers that use the $M \times V$ operation to produce object classes. With feature sizes of 4K floating point values and 1K object classes, the weight matrix $M$ can be a few MB in size. This is a prohibitive real-time computa-
vated by the above observation. Given the embedded nature such a memory system can provide computation-memory
provide the address for the memory access into array
to access elements of column indices, the values of
from
the memory itself could perform this metadata access in or-
Next, values from the sparse matrix row are read (shown in
vals (shown as
Gray blocks
vals)
vals
vals
This is 3
cols (blue blocks) are made first to obtain non-zero column indices. Next, using these column indices, the values of V are read (shown in yellow blocks). Next, values from the sparse matrix row are read from
vals (shown as green blocks). Finally, multiply-accumulate (MAC) operations are performed (shown as gray blocks) on values obtained from
vals] and V[]. This is 3 memory accesses per iteration for 1 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 in order to access 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 3. 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 Hardware Helper Thread accelerator is motivated by the above observation. Given the embedded nature of the systems, the HHT architecture is intended to satisfy the following requirements:

- **Fit into traditional programming model and leverage CPU-side features such as vectors**: Embedded cores have employed computational acceleration such as DSP instructions and vector extension. We envision the HHT as using this well-established programming model.
- **Accelerate the performance of in-order single-core, single-threaded applications**: Sensing devices often comprise just a single core. These devices do not possess sophisticated latency-hiding techniques such as multi-threading, out-of-order and speculative execution or non-blocking caches.

- **Integrate with a diversity of memory systems**: Embedded devices come in many sizes – some with only on-chip RAM while others possess a more sophisticated memory hierarchy comprising caches and DRAM. The HHT accelerator must be able to integrate with a wide variety of memory systems leveraging existing on-chip bus infrastructure and memory interfaces.

3. **DESIGN OF HHT**

In our work, we investigate a memory-side accelerator (HHT) that is either a dedicated ASIC hardware or a programmable unit using a simple RISC core. Figure 4 shows the system organization of a typical embedded system. It should be noted that embedded devices come in a range of configurations. At the low-end, an MCU is provisioned with a small amount of on-chip SRAM backed by external non-volatile storage (typically flash memory). On higher-end, embedded systems include a cache hierarchy followed by off-chip memory (typically DRAM). We envision our HHT to be either embedded or placed very close to the SRAM of an MCU or the L1D in higher-end systems as shown in Figure 4.

3.1 **ASIC HHT**

In the dedicated hardware version (referred to as ASIC HHT), the HHT is 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: buffer contains the matching elements of the vector in matrix-vector algorithms. The BE loads of matrix and vector metadata (associated with the sparse representation) 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.

![Diagram](image-url)
HHT. This programming is performed by writing to a set of memory-mapped registers (MMRs) in the FE. We list the MMRs needed to support the SpMV operation using CSR representation of sparse data.\footnote{We described ASIC HHT for SpMV here. The design can be extended for SpMSpV using additional metadata and comparing indexes of Matrix columns with Vector indexes to match non-zero values.} Values programmed into these configuration registers control the address generation and termination logic.

- \texttt{M\_Num\_Rows}: Number of rows of sparse matrix \( M \).
- \texttt{M\_Rows\_Base}: Base address of CSR rows array of \( M \).
- \texttt{M\_Cols\_Base}: Base address of CSR cols array of \( M \).
- \texttt{V\_Base}: Base address of dense vector \( V \).
- \texttt{ElementSizes}: Sizes for Rows, Cols, Vals arrays and Vector.
- \texttt{Start}: This bit is set last to trigger the hardware operation.

For the SpMV operation, the HHT provides \textit{indexed gather} support. Values from vector \( v[k] \) are gathered using indices from \texttt{M\_Cols} to construct buffers. The CPU performs vector loads of buffered values and multiply-accumulates into the output vector. Values collected into the buffers are read by the CPU via the normal load-store interface. In our design, we assume a vector-wide load-store interface for high-end embedded devices, but the HHT design can work with scalar load-store interfaces also.\footnote{In fact, our design works even better with scalar loads as there is less pressure on HHT to return a large number of values per loop iteration.} The software uses a \textit{fixed buffer address} to load from. Whenever the CPU performs a load, the FE updates its buffer state to determine when the buffer has been completely drained by the CPU. 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.

The FE is implemented with \( N \) vector-sized buffers where \( N \) is a design-time parameter. \( N >= 2 \) permits the HHT to prefetch and store buffers ahead of time. \( N = 2 \) provides double-buffer arrangement. The FE and BE work the memory pipeline managed by the control unit. Figure 5 describes the design of the HHT pipeline operation.

The first stage of the pipeline issues memory read requests to obtain contents of the \texttt{M\_cols[]} array. It uses the current array index stored in the register \texttt{cur\_col\_idx} to generate requests to the next \texttt{BLEN} elements where \texttt{BLEN} is the length of the buffer. In the next stage, memory response is stored in a \texttt{BLEN}-sized column-indices buffer. Values stored in this buffer are used to compute the addresses of the elements of array \( V[k] \). Given an index value \( k \) and vector element size \( s \), the address is computed as: \( V\_address = V\_Base + k \times s \). This computed address is used to issue a second memory request in stage 4 of the pipeline. Values read from array \( V[k] \) are stored in a CPU-side buffer. Depending on the number of buffers provisioned, the control unit tracks which buffer to write to.

The control unit generates signals for all stages of the pipeline. In particular, the unit tracks which buffer is the \textit{read buffer} – the buffer that the CPU will read from and which buffer is the \textit{write buffer} – the buffer that the HHT Back-End will fill. The control unit also tracks buffer empty/full conditions so as to stall CPU load requests (when no ready buffer is available), stall the memory request generation to \( V[k] \) (when column indices have not yet been read from memory) or to skip issuing new memory read requests when all buffers are full.

### 3.1.2 HHT Back-End

The HHT Back-End (BE) fetches metadata and data for the front-end. The BE uses the next vector of column indices to generate addresses for elements of \( v[k] \) and issue memory read requests. Address generation is straightforward: knowing the programmed base address of \( v[k] \) (stored in register \texttt{V\_Base}) and element size \( s \), for a column index \( k \), the element address is \( V\_Base + k \times s \). The BE works with the underlying memory system to issue read requests and to collect responses. 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. In the MCU integration, the BE issues requests to the on-chip RAM via an on-chip interconnect.

### 3.2 Programmable HHT

We also evaluate the use of a simple RISC-like processor for HHT. This HHT core will require very few integer instructions since HHT only deals with accesses to memory. Moreover, newer address modes are investigated that are better suited for handling sparse data representations, including but not limited to: use of two registers to specify address, indirect addressing, word-aligned addresses, bit-manipulation instructions (for bit-vector sparse representations). The primary CPU core and HHT core communicate through dedicated shared memory such as shared scratchpad to store metadata as well as buffers (see previous sections for descriptions of these items). Flags are used to indicate when a full buffer is released by HHT and when CPU consumes the buffer (spin locks are used to access these flags). We will describe the number of instructions needed by HHT to fill buffers using available RISC V instructions for SpMV algorithm in Section 4.
3.3 Other Considerations

Output Compression: It should be noted that the output of SpMV is likely to be dense. The output of SpMSpV, however, is likely to be sparse. We have configured HHT to track the position of a non-zero result produced from SpMSpV and store the value using a compressed format (CSR, for the purpose of this paper). Likewise, for convolution algorithms, we programmed HHT (both for ASIC and programmable implementations) to store output either in dense or sparse formats.

Saving and Restoring HHT State: The ASIC HHT state may be treated as part of the process state that is currently using the HHT. If the process is context-switched out, then the state of the ASIC HHT could be saved along with the CPU state (PC, stack, etc). The HHT state comprises the filled but unread buffers and the metadata bookkeeping registers (cur_row and cur_col_idx). The restoration of process state is similar: the unread buffers and state of the HHT metadata are restored. In case of a programmable HHT, CPU requests HHT core to save and restore its state (at the same time when CPU saves its state).

Precise Exceptions: We assume that the primary core provides proper metadata to HHT, (including the addresses of col, row and val arrays for CSR representations), and assume that accessing memory using these addresses would not cause memory exceptions. However, it is possible to design HHT to inform the primary CPU of any exceptions using a separate flag. For this contribution we will ignore any possibility of exceptions caused by HHT.

Multi-core Support: Our focus is embedded systems with a single core. However, a separate HHT can be associated with each core, accelerating matrix codes in multi-core environment.

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

3.4 Area and Power Estimates

The area of ASIC HHT is a sum of the logic gates of the control unit and storage for pipeline stages, HHT buffers, memory-mapped registers, internal state registers and column-index storage. We designed ASIC HHT for CSR sparse representations as described earlier using System Verilog and ARM libraries with Synopsys Design Compiler- 2020.09 tool for design synthesis. We synthesized three different feature sizes (28nm, 16nm and 7nm) running at four different clock speeds (0.375 GHz, 0.85GHz, 1.1 GHz and 1.4 GHz). For this contribution we will use 28nm feature sizes and 0.375 GHz clock. Our HHT is approximately less than 1% the size of a PULP v1-like RISCV core [44]. The HHT consumes 0.793 mW power while the RISCV core consumes 115.4 mW power.

4. PROGRAMMING MODEL

HHT need to be configured with matrix metadata before the start of SpMV, SpMSpV, convolution or other computational kernels. This is done by programming memory-mapped registers in the HHT. These registers provide metadata information such as matrix dimensions, base addresses of CSR arrays, base address of the vector, etc. (or metadata for other sparse representations) as described in Subsections 3.1.1 and 3.2.

4.1 Accessing Data Buffers in the SPMV Kernel

The HHT data buffer is memory-mapped to a fixed address (or shared scratchpad). Note that even if the HHT uses multiple physical buffers for “double buffering” operation, the CPU software accesses the current buffer via the same address. In this sense, the HHT 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 vector-load from the fixed address. The HHT routes this load request to the correct buffer and returns load data.

Putting it all together, Figure 6a compares the traditional software-based SpMV code with the HHT-based implementation shown in Figure 6b.

For simplicity of illustration, both codes are shown using scalar implementations. The code on the left is the traditional SpMV where the indirect access V[M_cols[i]] is performed in each iteration of the inner loop. The code outlined on the right is the HHT-accelerated version. Relevant changes are highlighted as blue text. The volatile pointer variable BUFFER is initialized to the address of the HHT buffer. Next, before executing the main kernel, the HHT is initialized inside the HHT_Init call. This function is supplied with necessary arguments and the function implementation (not shown for brevity) is simply a series of writes to HHT configuration registers (or to shared scratchpad). The main kernel of the HHT-version looks very similar to the software-only version. The only change is the way that values from V are accessed. In the HHT-version, these values are accumulated into buffers by the HHT and software reads them from the buffer using the BUFFER pointer.

Figure 6a also includes the RISCV code that will be executed by programmable implementation of HHT. This code (shown in a box) executes to obtain V[M_cols[i]]. If a new load instruction of the type lw R-destination,R-index, R-offset were implemented in RISCV, we can replace add a5,a3, a5 and lw a5, 0(a5) with lw a5, a3, a5). Likewise the last two instructions can be combined into a single lw a5, a4, a5. We show the performance results for both cases. We feel that additional instructional optimizations are possible if a RISC-like core is used for HHT.

5. EXPERIMENTAL EVALUATION

We evaluate both ASIC HHT and programmable HHT for embedded processing environments. We use the Spike [20] simulator representing an embedded CPU core (as well as a RISCV core for programmable HHT).

3In our experiments, we mapped the HHT buffer to address 0x3000_2000.

4The volatile attribute ensures that each read goes to memory.
void spMV(void) {
    int s;
    int k=0;
    for(int i=0;i< M_num_rows;i++) {
        int nnz = M_rows[i+1]-M_rows[i];
        s=0;
        for(int j=0;j<nnz;j++) {
            s+=M_vals[k+j]*v[M_cols[k+j]];
        }
    }
}

volatile int * BUFFER=&HHT_BUFFER;
void HHT_MV(void) {
    int s;
    int k=0;
    HHT_Init(M_num_rows, M_rows, M_cols, V);
    for(int i=0;i< M_num_rows;i++) {
        int nnz = M_rows[i+1]-M_rows[i];
        s=0;
        for(int j=0;j<nnz;j++) {
            s+=M_vals[k+j]*(*BUFFER);
        }
    }
}

(a) Traditional Software version of \texttt{spMV}
(b) HHT-accelerated version of \texttt{spMV}

Figure 6: Comparison of \texttt{spMV} codes

Table 1: System Configuration

<table>
<thead>
<tr>
<th>Processor</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>RISC32 with IMAPDCV Extensions Frequency = 1.1 GHz Vector width (VL) = 8 Elements Element Size (SEW) = 32 bit Vector Arithmetic Latency = 4 cycles</td>
</tr>
<tr>
<td>ASIC HHT</td>
<td>No2 Buffers Buffer size = 32B</td>
</tr>
<tr>
<td>Programmable HHT</td>
<td>L1-Cache = 1KB Scratchpad Memory = 2KB</td>
</tr>
<tr>
<td>RAM</td>
<td>Size = 1MB</td>
</tr>
</tbody>
</table>

5.1 System Configurations

Table 1 describes the system configuration used in our experiments. The system includes a 32-bit RISCV [16] base architecture along with vector, compressed, atomic, multiply, floating and double precision extensions. The primary CPU core uses an in-order 3 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 buffers and RAM as shown in Figure 4. A simple in-order 32-bit integer RISCV with reduced instructions and 1KB instruction and 2KB data caches is used for programmable \textit{HHT}. ASIC \textit{HHT} is equipped with a 32B buffer to communicate with the primary CPU core.

We used \texttt{spike} [20] simulator for our work. We configured \texttt{spike} to match our design as described in Section 3. We incorporated several extensions to the baseline \texttt{spike} simulator including multi-cycle instruction latency, RAM memory model and processor wait cycles. Our extensions provide for cycle-accurate simulation environment. We collected total execution cycles, the number of cycles the CPU (primary RISCV core) is waiting for \textit{HHT} to fill buffers and the number of cycles \textit{HHT} is waiting for CPU to release free buffers.

5.2 Workloads

To analyze the performance of our accelerator carefully, we generated synthetic matrices of different sizes and different sparsity levels. We included several matrices drawn from the Texas A&M Sparse Matrix collection [12]. While this collection was originally intended for studying scientific codes, we selected matrices that are small enough in size that they are comparable to matrices encountered in DNNs. Furthermore, scientific workloads often exhibit greater sparsity than DNNs thereby evaluating the \textit{HHT} on a more complete range of real-world sparse matrices. We also used several matrices drawn from trained DNNs. Due to space limitations, we will only show a subset of the results for selected set of workloads, but describe our observations for other datasets. Since the sparsity of DNN datasets vary from network to network and layer to layer, our exploration using randomly generated inputs can provide an estimate of potential performance gains for different DNNs.

6. RESULTS

We present results for ASIC implementation of \textit{HHT} as well as for programmable implementation of \textit{HHT} using a simple RISCV processor. As described in Section 3, in case of a dedicated hardware, we rely on a pipelined implementation of \textit{HHT} which can generate one element of the Vector (in Matrix-Vector algorithms) per cycle. For programmable implementation, we present results using conventional RISCV instructions for obtaining Vector values corresponding to non-zero values of (rows) the matrix, as well as using optimized RISCV instructions as described in Section 4. We present results using one buffer and two buffers (double buffering).

We show performance gains (speedup over a baseline that uses a single CPU that performs both index computations and matrix-vector multiplications). We also present the fraction of time CPU is waiting for \textit{HHT}. Our goal is to offload “work” to \textit{HHT}, and overlap this work with that of the CPU. In the ideal case, CPU should not be waiting for \textit{HHT} when such an overlap is achieved. However, if \textit{HHT} is assigned larger share of “work”, CPU is likely to be idle. Thus, it is necessary to carefully evaluate the amount of work that is offloaded to \textit{HHT} to achieve performance gains.

We first present the results for sparse Matrix - dense Vector (\textit{SpMV}) multiplication, using randomly generated matrices
with varying degrees of sparsity (% of zeros). We will then present results for sparse Matrix - sparse Vector ($SpMSpV$) multiplication, again using randomly generated matrices and vectors with varying degrees of sparsities. Finally, we will present results for convolution kernels using randomly generated data. We expect that the use of random inputs with different sparsities provide valuable insights on the range of performance gains achieved with $HHT$.

### 6.1 HHT On SpMV Workloads

To understand the impact of sparsity on $HHT$ performance, we evaluated the $HHT$ on synthetic matrices with varying degrees of sparsity. Figure 7 presents the performance improvement achieved by $HHT$ over the baseline using only one CPU on a 512 * 512 matrices with sparsity varied from 10% to 90% when executing sparse Matrix - dense Vector ($SpMV$) multiplication. The application CPU is a RISCV core with vector extensions (in our experiments with 8-wide vectors). The figure contains 6 bars for each sparsity level: the left two bars (labeled Dedicated_HHT_1buffer and Dedicated_HHT_2buffer) represent the speedup achieved using an ASIC $HHT$ with 1 and 2 buffers.

The next two bars (labeled Programmable_HHT_1buffer and Programmable_HHT_2buffer) represent the speedup using a programmable $HHT$ with conventional RISCV instructions, for 1 and 2 buffers.

The last two bars (Optimized_Programmable_HHT_1buffer and Optimized_Programmable_HHT_2buffer) represent the speedup when $HHT$ core uses optimized instructions as outlined in Section 3.2. All results are for the case where the primary CPU core is using 8-wide vectors and vector instructions.

It can be seen from the first set of bars (i.e. labelled as Dedicated_HHT_1buffer in Figure 7), using dedicated hardware, $HHT$ consistently outperforms baseline with an average speedup of 1.70 (speedups range from 1.67 to 1.72) for ($SpMV$). The gains are smaller at higher sparsities, since the amount of work that is offloaded to $HHT$ depends on the sparsity of data – at higher sparsities, $HHT$ is assembling fewer data items for consumption by the CPU. As stated in Section 6, the performance gains depend on the amount of work that is offloaded to $HHT$. The second set of bars (i.e. labelled as Dedicated_HHT_2buffer in Figure 7) shows the speedup with 2 buffers, which shows an average speedup of 1.73 over the baseline and speedup ranges between 1.71 to 1.75. In general double-buffering helps in reducing the CPU wait times, which will be discussed shortly in Section 6.3 and improves speedup. However, the minimal improvement over one buffer is because, in case of an ASIC $HHT$, the CPU is not waiting for $HHT$ when a traditional RISCV core is used for $HHT$. We see very minimal performance gains, averaging 2% (third and fourth set of bars labeled Programmable_HHT_1buffer and Programmable_HHT_2buffer in Figure 7). However, when an optimized instruction set is used (see Section 3.2), we see an average of 23% speedup (last two sets of bars labeled Optimized_Programmable_HHT_1buffer and Optimized_Programmable_HHT_2buffer in Figure 7). The use of 2 buffers with programmable $HHT$ implementations results in higher performance gains: two buffers reduce the amount of time CPU is waiting for $HHT$ (more details in Section 6.3).

### 6.2 HHT On SpMSpV Workloads

Figure 8 shows the performance gains achieved by $HHT$ for sparse Matrix - sparse Vector ($SpMSpV$) multiplication. Similar to the data presented in Figure 7, we include six bars of data, three using a single buffer and three using two buffers. The first two bars (Dedicated_HHT_1buffer_1variant and Dedicated_HHT_2buffer_1variant) represent the speedups using an ASIC $HHT$, with one and two buffers respectively, and they show an average speedup of 2.47 times (speedup ranging between 1.48 times to more than 4.0 times). The speedup increases with sparsity, since at higher sparsities, there are fewer matching values to supply. In these experiments, our $HHT$ provides matching pair (or “aligned values”) of non-zeros from the sparse Matrix and the sparse Vector; the application CPU multiplies the pairs of values and accumulates the products. Thus, $HHT$ is performing more work than the CPU as it needs to traverse through (row) indexes of non-zero values in the Matrix and indices of the non-zero values of the Vector, and select values if the corresponding indexes match. ASIC $HHT$ requires more complex hardware. In the case of a programmable RISCV core for $HHT$, supplying matching non-zero values requires the execution of more instructions (and more cycles). In such situations, CPU will be idling for longer periods of time, waiting for $HHT$ - we will describe the amount of time CPU waits for $HHT$ in Section 6.3. To balance the work between $HHT$ and primary CPU, $HHT$ can be tasked to only supply the Vector values (either a non-zero value if there is non-zero element at the matching index or a zero). The CPU is responsible for fetching non-
zero values of (a row) the Matrix, and use the Vector values supplied by HHT to compute the results. In Figure 8, the next two bars (labeled Dedicated_HHT_1buffer_2variant and Dedicated_HHT_2buffer_2variant) represent results where ASIC HHT only supplies Vector values (with one and two buffers). As can be observed, this variant HHT performs much better than baseline (CPU only) and better than when HHT supplies both matrix and vector values, at lower sparsities. At higher sparsities, greater than 80%, this variant performs worse than first HHT variant but better than baseline CPU version. This is because, at higher sparsities, the CPU is supplied with more zero values of Vector (when there is no matching Vector element corresponding to non-zero Matrix values) and these zero computations should be considered as wasted computations. On average, this version of ASIC HHT performs 3.05 times better than the baseline (speedup range between 2.5 times and 3.52 times). The last two bars (labeled Programmable_HHT_1buffer_2variant and Programmable_HHT_2buffer_2variant) show the results when a programmable RISCV core is used to provide matching Vector values. We see a speedup of 2.67 times over the baseline (speedup range between 2.25 and 3.05 times). As expected, programmable HHT has lower speedups than ASIC HHT. In general programmable HHT benefits more from two buffers than an ASIC HHT. Since ASIC HHT is fast enough to provide buffers to CPU, using 2 buffers does not significantly improve performance. However, in programmable case, as the CPU is waiting for HHT, and if HHT has to wait for the single buffer to be read by CPU before filling it again, the delay can be excessive. However when we use 2 buffers, HHT starts filling the second buffer while CPU is reading the first buffer.

6.3 CPU - HHT Overlap

Our goal is to offload computations (or work) related to metadata processing to locate row-column indices of sparse data to the HHT and overlap this work with multiply-accumulate operations on CPU side. The performance gains depend on the amount of work offloaded. However, if too much work is offloaded to HHT, CPU may be waiting for HHT. In this section, we will analyze the fraction of the time CPU is waiting for HHT (or the amount of time CPU is idling).

Figure 9 shows the fraction of time CPU is idling (waiting for HHT) for sparse Matrix-dense Vector (SpMV) multiplication. The figure includes six bars for each sparsity level, and these bars parallel the bars shown in Figure 7. The first two bars represent the fraction of the time CPU is idling when an ASIC HHT is used with one and two buffers. The next two bars show the idle times when a conventional RISCV ISA is used for programmable HHT (with one and two buffers) and the last two bars show the data using an optimized instruction set for programmable HHT. With an ASIC HHT, the application CPU rarely waits. This also results in maximum speedup as shown in Figure 7. When conventional RISCV ISA is used, CPU idles as much as 40% of the time. This wait fraction is reduced significantly when an optimized RISCV instruction set is used for programmable HHT. In almost all cases, the idle time is reduced with two buffers.

Figure 10 shows the CPU idle times for sparse Matrix - sparse Vector multiplication (SpMSpV). The six bars for each sparsity level parallel the six bars shown in Figure 8 (i.e., first two bars are for ASIC HHT that provides matching non-zero values of Matrix and Vector with one and two buffers; the next two bars are for ASIC HHT that only provides non-zero values of the Vector and the last two bars are for programmable HHT that only provides matching Vector elements). The results are very similar to those for SpMV computations shown in Figure 9. When HHT is supplying aligned Matrix and Vector non-zero values, CPU is idling for a significant fraction of the total execution time. Two buffers show only minor improvements. When HHT only provides Vector elements, the CPU idle times are significantly reduced, and two buffers improve idle times greatly. Programmable HHT causes more CPU idle times than ASIC HHT.

6.4 Sensitivity To Vector Widths

Results shown thus far are obtained using RISCV vector instructions with vector width of 8. To understand the compute-memory overlap and the benefit of HHT, we experimented with different vector widths used in the vector instructions of RISCV: 1, 4 and 8. We did not consider larger vector widths as they would require higher memory bandwidth and area – both are significant constraints in embedded systems.

Figure 11 plots the improvement achieved by HHT over correspondingly sized vector versions of sparse Matrix -
dense Vector (SpMV) multiplication using CPU-only baseline. The figure includes six bars for each sparsity (for a 512*512 sparse matrices). We show performance gains using ASIC HHT over baseline and performance gains achieved using a programmable HHT for three different vector widths, 1, 4 and 8. The first two bars are for vector width of 1 (scalar), next two are for vector width of 4 and the last two bars are for vector width of 8. While the ASIC HHT maintains high levels of speedup for all vector widths (Speedup ranges between 1.77 - 1.81 for scalar, 1.51 - 1.62 for vector width of 4, 1.71 - 1.75 for vector width of 8), the programmable HHT loses its performance gains as the vector widths increases (Speedup ranges between 1.77 - 1.81 for scalar, 1.49 - 1.55 for vector width of 4, 1.24 - 1.22 for vector width of 8). This is expected since programmable HHT has to assemble several non-zero values (of the Vector in SpMV) while the primary CPU core consumes all those values using vector arithmetic. This is inline with our observation that the performance gains depend on the amount of work offloaded to HHT and if this work can be overlapped with that of the primary core. As previously described in Section 6.3, CPU idles significant fraction of the time when using vector widths of 8.

6.5 Energy Savings

Using the power estimates described in Section 3.4 for 28nm at 0.375GHz clock, we calculated energy savings when the ASIC HHT is used on a 512*512 sparse Matrix - dense Vector multiplications (i.e., SpMV). We use a scalar PULP v1 like RISCv core for the application CPU. We compare the energy consumption with that of a baseline RISCv core that performs all operations. On average, the use of HHT results in 55% reduction in energy consumed for SpMV across the different sparsities, when compared with baseline.

6.6 CPU-HHT Versus Two CPU Cores

It should be noted that even when a simple RISC-like core is used for HHT, it should not be considered as a second full-fledged core. If two identical cores are used and the matrix-vector multiplication is multi-threaded on these two cores, the maximum speedup one can expect is two. We collected speedup data on two Intel-based processing cores, and created two threads using OpenMP for CSR-based SpMV computations. The cores did not use vector instructions. Using the same randomly generated 512*512 matrices we used in our experiments described in Section 6.4, the two core implementation shows speedups ranging between 1.47 and 1.81 over a single core implementation. The less than ideal 2 times speedup is because of parallel overhead and load imbalance due to the random distribution of nonzero values among rows of a matrix. As can be seen from Figure 11, our programmable HHT (with scalar CPU) with 2 buffer configuration achieves comparable performance as two cores ranging between 1.77 - 1.81 with an average speedup of 1.81. Moreover, the RISC core for HHT can be designed with very few instructions, fewer registers (for example 16 integer registers) and very small amounts (1-2K Bytes) of cache memories, leading to energy savings compared to two full-fledged cores. We have already outlined some optimizations in terms of instructions in Section 3.2 and Section 4. In the future, we will explore the implementation of such simple cores that can be used to process a wide-variety of sparse data formats.

6.7 HHT on Fully-Connected Layers of DNNs

The fully connected layer of DNNs performs Matrix-Vector multiplication before the final classification. We leveraged the quantized weights matrix of this layer from a variety of networks: MobileNet [29], MobileNetV2 [46], DenseNet [30], ResNet [27], ResNetV2 [28], and VGG16, & VGG19 [48]. Figure 12 plots the performance improvement achieved by ASIC HHT over the baseline for these workloads. Baseline (using CPU only) SpMV codes use the RISCv vector extensions to process VLEN elements in parallel (VLEN = 8 in our setup). The baseline also uses the vector indexed-load instruction to gather values using indices specified in a vector register. This instruction is similar to Intel AVX2 Gather instruction [31].

ASIC HHT achieves speedup over the baseline sparse version of SpMV anywhere from 1.53 times on DenseNet to 1.92 times on VGG19. The performance improvement running DNN data sets are similar to the synthetic results at different sparsity and matrix sizes.

6.8 HHT on Sparse Convolutions

We used CSR-based representation for each channel of the input features (and dense weights) to compute convolutions. The convolution layer repeatedly performs a dot-product of the same set of filters over different windows of an input matrix. The ReLU activation function is performed element-wise over an entire matrix, returning zero if a value is negative and the value itself otherwise. Finally, the max pooling layer iterates through each 2x2 block of an input matrix and returns

Figure 11: HHT Sensitivity to Vector Width

Figure 12: HHT on DNN workloads

The ASIC HHT achieves even higher performance gains.
only the maximum element in a block\textsuperscript{7} [50].

We explored combining activation and pooling layers with convolution layers to minimize memory accesses, since performing activation and pooling sequentially after convolution requires multiple accesses to the results produced by the convolution layer. The labels \textit{sddap} and \textit{sdsap} indicate the storage formats for the inputs, weights, and outputs of the convolution layer. The label \textit{sddap} refers to experiments where the input is in sparse representation \((s)\), the weights are dense \((d)\), and the output of the convolution is stored in dense format \((d)\); then activation \((a)\) followed by pooling \((p)\) are combined with convolution layer. Likewise, \textit{sdsap} indicates that the input is in sparse representation, weights are in dense representation and output is stored in sparse representation. The advantage of \textit{sdsap} is that the result of one layer is already stored in sparse format to feed as input to the next layer in the network.

We randomly generated inputs with different sparsities (the input image size is set to 28*28, which is the same as images from the MNIST dataset [13, 34]). As shown in Figure 13 for input matrices with sparsity up to 80%, the ASIC \textit{HHT} implementation for both \textit{sddap} and \textit{sdsap} achieve performance gains over their respective CPU-only implementations. For sparsity under 50%, we see 2-2.5 times speedup, and for sparsity between 50% and 75%, we see a speedup of about 1.5 times. When the sparsity is above 80%, the reduction in computations resulting from higher levels of sparsity in the CPU-only version outweigh the gains from offloading the metadata computation to the \textit{HHT} accelerator. This is because the \textit{HHT} implementation for the convolution algorithm expands each window of the sparse input features, filling the buffer with zeros when necessary, thus forcing the CPU to perform multiplications with zeros.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{speedup_achieved_by_asic_hht_for_convolution.png}
\caption{Speedup Achieved By ASIC \textit{HHT} For Convolutions}
\end{figure}

Since different DNN datasets exhibit different levels of sparsity and the sparsities vary from layer to layer, we feel that our study can be helpful in estimating the performance gains of using \textit{HHT}. We analyzed the average sparsity across all layers of four state-of-the-art convolutional neural network models (VGG16 & VGG19 [48], ResNet [27], and MobileNetV2 [46]). Table 2 shows that the input features of most layers have a sparsity below 80% with the average layer sparsity falling in the range of about 40%-60%, not counting layers with fully dense input features (0% sparsity). This would translate to performance gains of 1.5-2 times on average if ASIC \textit{HHT} is used instead of CPU-only, for both \textit{sddap} and \textit{sdsap}.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Dataset & VGG16 & VGG19 & ResNet & MobileNetV2 \\
\hline
Average & 62\% & 59\% & 57\% & 37\% \\
Maximum & 92\% & 92\% & 85\% & 87\% \\
\hline
\end{tabular}
\caption{Sparsity Statistics for CNN Datasets}
\end{table}

\subsection{6.9 HHT on Scientific Workloads}

We evaluated the \textit{HHT} with a large number (over 1000) of scientific matrices from the TAMU collection [12]. Data for scientific applications exhibit high-levels of sparsity (for the matrices we tested, the average sparsity was 98\%). On average, the ASIC \textit{HHT} outperforms baseline implementation of \textit{SpMV} on these workloads by 24\% (or a speedup of 1.32).

\section{7. RELATED WORKS}

\textbf{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 [54], accelerator for very large \textit{SpMV} [45]. The work in [45] proposes a Two-Step \textit{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 memory bottleneck faced by vectorized codes running on traditional cores. Some researchers explored hardware that expands sparse data into dense by inserting zeroes [6], [1]. It is believed that at lower sparsities, such expansion can improve performance since the expanded data can be executed using vector instructions.

There are several works that attempt to improve the performance of sparse matrices for scientific applications. Authors of [11] proposed a parallel sparse matrix algorithm based on SUMMA used in BLAS library and parallelized the sparse matrix multiplication, while we used hardware helper to extract only non-zero values to CPU. Greathouse [22] 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 [3] proposed a parallel sparse Matrix - sparse Vector (\textit{SpMSpV}) algorithm that stores the product of sparse Matrix - dense Vector based on the row indices and later accumulates it, all by using buckets.

\textbf{Accelerators for Machine Learning.} Interest in DNN based accelerators have seen a rise in recent years. There are too many different hardware/software implementations 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 multiplications. Sparse data is represented as bit-vectors and

\textsuperscript{7}Average pooling works similarly, except doing an average rather than finding the maximum.
dataflow like Multiply-Accumulate units are configured based on the non-zero values in data. Moreau, et.al. [36] 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.

Processing In Memory and Near Data Processing Approaches. There have been many studies on near-data processing (or Processing-In-Memory) logic. 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 [14, 43, 52]). 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 [18].

New Sparse Representations. In a different vein, there have been proposals on improving compression of sparse matrices and proposed techniques including hierarchical bit vectors [32] or compression on top of CSR [42]. There are proposals for specialized hardware to compress and decompress data for use by CPU (assuming that the CPU uses conventional SpMV software) [42]. Others propose hardware to use the new compression formats (such as hierarchical bit maps) for performing sparse matrix computations [32]. We programmed HHT to handle sparse data represented using SMASH [32] format. SMASH format requires complicated indexing to locate the row and column positions of non-zero values of a sparse matrix. This implies that HHT is performing more work that 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 details about the performance gains achieved when HHT is programmed to process hierarchical bit representation of sparse data as done in SMASH [32].

8. CONCLUSIONS

In this work, we presented a memory-side accelerator to accelerate sparse matrix-vector and convolution computations. The accelerator, denoted as Hardware Helper Thread or HHT, removes the overhead of accessing and interpreting sparsity metadata from the primary CPU core. We evaluated using both a dedicated or ASIC hardware for HHT, as well as using a simple RISCV core as programmable HHT so that it can handle different sparse formats. Our approach should be distinguished from other accelerators that accelerate the entire computation, not just index computations.

ASIC implementation of HHT can only deal with a specific sparse representation and a specific matrix computation. However, ASIC HHT produces higher speedups than programmable HHT. We evaluated the use of our HHT (both ASIC and programmable versions) for sparse Matrix-dense Vector (SpMV), sparse Matrix - sparse Vector (SpMSpV) multiplications as well as convolution algorithms. The use of HHT offloads some of the index computations needed for sparse data representation to HHT and these computations are overlapped with the computations of the primary CPU core. We evaluated our designs to augment RISCV cores with vector instructions that are designed for embedded system. We present the performance gains achieved by our designs over baseline implementations of a CPU core that executes all computations. We used synthetic matrices with different sparsities (i.e., fraction of values that are zeros). The performance gains depend on the amount of work offloaded to HHT and the amount of time the primary CPU is waiting (or idling) for HHT to provide data. ASIC HHT shows significantly higher speedups over the baseline than the programmable HHT, since the HHT is able to provide needed data to CPU without any delays. The ASIC HHT shows average performance gains ranging between 1.7 and 3.5 depending on the sparsity levels, vector-widths used by RISCV vector instructions and if the Vector (in Matrix-Vector multiplication) is sparse or dense. The programmable HHT shows average performance gains ranging between 1.23 and 1.8. In case of programmable HHT implemented using a simple RISCV core, HHT cannot always keep up with CPU’s demand for data. CPU wait times (or idle cycles) can be reduced to some extent by using double-buffering. We also show energy savings of 55% when ASIC HHT is used compared to baseline (for SpMV).

Although not included in this contribution due to space limitations, we have evaluated the use of our HHT (both dedicated and programmable versions) to process other sparse data representations including bit-vectors (where the position of a nonzero value is represented by a "1" for the corresponding bit of the bit vector) and a hierarchical bit vector representation called SMASH [32]. SMASH format requires complicated indexing to locate the row and column positions of non-zero values of a sparse matrix. This implies that HHT is performing more work that the CPU, causing CPU to idle. Moreover, we feel that SMASH formats may not be suitable for embedded systems.

We feel that the RISCV core needed for programmable HHT can be even simpler than traditional 32-bit integer RISCV. It can be designed with very few integer instructions, very small instruction and data caches, thus requiring smaller silicon area and consuming less energy than a full-fledged primary CPU core. We are currently exploring the design of such RISCV which can be programmed to handle many different sparse representations.

REFERENCES


