Heterogeneity in Computing Workshop, held in conjunction with IPDPS 2022, May 30, 2022, Lyon, France

HETEROGENEOUS ARCHITECTURE FOR SPARSE DATA PROCESSING

Shashank Adavally, Alex Weaver, Pranathi Vasireddy,
Krishna Kavi, Gayatri Mehta, Nagendra Gulur

University of North Texas
Denton, Texas, USA

ABSTRACT

Sparse matrices are very common types of information used in scientific and machine learning applications including deep neural networks. Sparse data representations lead to storage efficiencies by avoiding storing zero values. 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.

In this paper, we will explore a dedicated hardware for a memory-side accelerator called *Hardware Helper Thread* (*HHT*) that performs all the necessary index computations to fetch only the nonzero elements from sparse matrix and sparse vector and supply those values to the primary core, creating heterogeneity within a single CPU core. We show both performance gains and energy savings of *HHT* for sparse matrix-dense vector multiplication (*SpMV*) and sparse matrix-sparse vector multiplication (*SpMSpV*). 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. We also show energy savings of 19% on average when ASIC *HHT* is used compared to baseline (for *SpMV*), and the *HHT* requires 38.9% of a RISCV core area.

Index Terms— 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 [1–3]. 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 [4–6].

Sparse matrix-vector multiplications are at the heart of machine learning, data analysis and scientific applications. Algorithms such as forward and back-propagation in deep neural nets [7], graph neural nets [8], Markov clustering [9], high-dimensional similarity search [10], topological similarity search [11], clustering coefficients [12], betweenness centrality [13], multi-source breadth-first-search [14], label propagation [15], and solvers of discretized differential equations [16] 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 multiplication, (denoted as *SpMV*) and sparse matrix and sparse vector multiplication(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 [17]), block compressed CSR (BCSR [18]), compressed sparse column (CSC [19]), coordinate list (COO [20]), bit-vectors [5], and run-length encoding [5]. There are also some newer representations including hierarchical bit vectors [21] and compression on top of CSR [22]. 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.

It has been documented that accessing and processing compressed *metadata* incurs significant overheads [23]. 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 paper, we describe the design and evaluation of such a memory-side hardware called *Hardware Helper Thread* or *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 a dedicated (or ASIC) implementation of *HHT*.

2. We evaluate *HHT* for both sparse Matrix - dense Vector (*SpMV*) and sparse Matrix - sparse Vector (*SpMSpV*) multiplications.

3. We provide detailed design and implementation of our dedicated hardware *HHT* for handling CSR representation.We analyze the silicon area needed, estimated power consumed and the energy savings achieved by our *HHT* augmenting a RISC V core executing *SpMV* computations when compared to a baseline RISCV core executing all computations.

Our work should be contrasted with prefetching works, including recent study that describes a programmable prefetcher [24] which can be programmed to prefetch based applications access patterns. Our *HHT* brings **only needed** data to the primary core. Our focus in the paper is only on accelerating index accesses unlike other accelerators that aim to accelerate entire computational kernels, such as the numerous neural network accelerators that perform all computations involved in complex networks.

2. MOTIVATION FOR ACCELERATING INDEX COMPUTATIONS

Consider the *SpMV*algorithm that multiplies a sparse matrix M by a dense vector V to produce an output (dense) vector Y. Figure 1 shows a sample 3×3 matrix M using compressed sparse rows (CSR) representation.

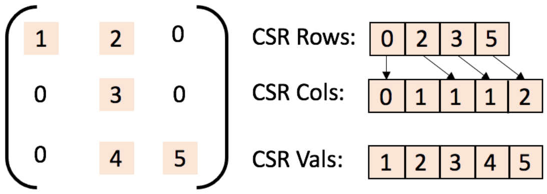

Fig. 1: A 3x3 sparse matrix in CSR and Bit-Vector Formats

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.

The *SpMV*algorithm traverses M row by row, obtains the column indices of the NZ values, and accesses the corresponding indices of the (dense) vector V. An outline of this algorithm implemented for a CSR representation of M is shown in Algorithm 1.

Algorithm 1 CSR Version of *spMV*

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

Among the memory accesses made by this code, the indirect accesses performed by $v[cols[.]]$ are expensive – these indirect accesses require accessing $cols[.]$ before values of $v[.]$ can be read. Processor vendors have offered vector gather instructions (see Intel [25], ARM [26] and RISCV [27]) 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 can not 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*. A more detailed analysis of the overhead was included in [23]. 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 is the motivation for the design of a heterogeneous CPU-*HHT* core.

For this publication, 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 lower 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 (see Fig. 2). Thus, achieving intelligence at the edge requires highly optimized implementations of various types of ML inference algorithms.

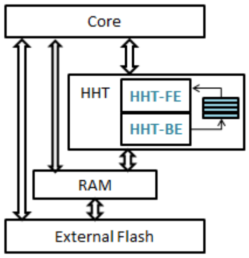

Fig. 2: HHT in Microcontroller Systems

3. DESIGN OF HHT

In our work, we investigate a dedicated ASIC memory-side accelerator (*HHT*). We envision our *HHT* placed either inside microcontroller memory (typically SRAM based memory) or very close to the memory. When a microcontoller contains cache memories, *HHT* will access the cache for fetching sparse data.

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.

3.1. HHT Front-End

The *HHT* FE is responsible for matrix metadata configuration, and coordination with the CPU. The FE has to be configured by software to store matrix metadata into the *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.[1] Values programmed into these configuration registers control the address generation and termination logic.

- M_Num_Rows: Number of rows of sparse matrix M.
- M_Rows_Base: Base address of CSR rows array of M.
- M_Cols_Base: Base address of CSR cols array of M.
- V_Base: Base address of dense vector V.
- $ElementSizes$: Sizes for Rows, Cols, Vals arrays and Vector.
- $Start$: This bit is set last to trigger the hardware operation.

For the *SpMV* operation, the *HHT* provides *indexed gather* support. Values from vector $v[.]$ are gathered using indices from 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.[2] The software uses a *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 3 describes the design of the *HHT* pipeline operation.

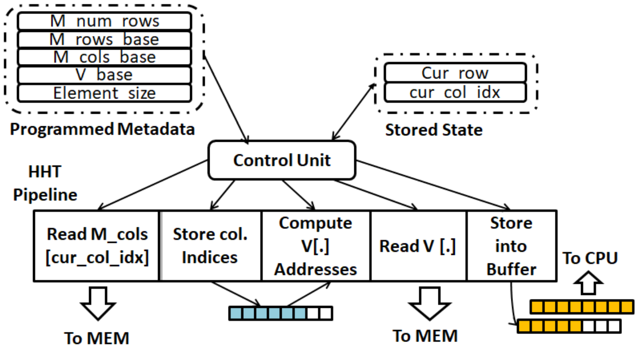

Fig. 3: *HHT* Pipeline

The first stage of the pipeline issues memory read requests to obtain contents of the $M\_cols[.]$ array. It uses the current array index stored in the register cur_col_idx to generate requests to the next $BLEN$ elements, one element at a time, where $BLEN$ is the length of the buffer [3]. In the next stage, memory response is stored in a $BLEN$-sized column-indices buffer. Values stored in this buffer are used to compute the addresses of the elements of array $V[.]$. Given an index value k and vector element size s, the address is computed as: $V\_address = V\_Base + s \times k$. This computed address is used to issue a second memory request in stage 4 of the pipeline. Values read from array $V[.]$ are stored in a CPU-side

[1] 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.

[2] 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.

[3] This corresponds to vector width used by the RISCV vector instructions.

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 *read buffer* – the buffer that the CPU will read from and which buffer is the *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 $V[.]$ (when column indices have not yet been read from memory) or to skip issuing new memory read requests when all buffers are full.

3.2. 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 $v[.]$ and issue memory read requests. Address generation is straightforward: knowing the programmed base address of $v[.]$ (stored in register 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 a 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.

4. EXPERIMENTAL EVALUATION

We evaluate ASIC *HHT* for embedded processing environments. We use the *Spike* [28] simulator representing an embedded CPU core.

System Configuration: Table 1 describes the system configuration used in our experiments. The system includes a 32-bit RISCV [29] 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 and stall the pipeline. The vector unit is not pipelined. The memory comprises buffers and RAM.

We configured *spike* [28] to match our design as described in Section 3. We incorporated several extensions to the baseline *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 *HHT* to fill buffers and the number of cycles *HHT* is waiting for CPU to release free buffers.

Workloads: To analyze the performance of our accelerator carefully, we generated synthetic matrices of different sizes and different sparsity levels. We also analyzed *HHT* using several matrices drawn from the Texas A&M Sparse Matrix collection [30]. However, due to space limitations, we did not include those results here. The speedup results are inline with those for synthetic workloads presented in this paper - noting that Texas A&M Sparse Matrix data has very high sparsity levels (greater than 90%). Since the sparsity of DNN datasets varies 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.

Table 1: System Configuration

Processor	Values
Core	RISCV ISA with 32 bit Floating-point Extensions Frequency = 1.1 GHz Vector width (VL) = 8 Elements Element Size (SEW) = 32 bit Vector Arithmetic Latency = 4 cycles
ASIC *HHT*	N=2 Buffers Buffer size = 32B
RAM	Size = 1MB

5. RESULTS

We show performance gains achieved by *HHT* over the 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 *HHT*. Our goal is to offload ”work” to *HHT*, and overlap this work with that of the CPU. In the ideal case, CPU should not be waiting for *HHT*. However, if *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 *HHT* to achieve best performance gains.

We first present the results for sparse Matrix - dense Vector (*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 fully connected layers of several Deep Neural Net architectures. We expect that the use of random inputs with different sparsities provide valuable insights on the range of performance gains achieved with *HHT*.

5.1. HHT On SpMV and SpMSpV Workloads

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 4 presents the performance improvement achieved by *HHT* over the baseline using only CPU on a 512 * 512 matrix with sparsity varied from 10% to 90% for sparse matrix - dense Vector (*SpMV*) multiplication. The primary CPU is a RISCV core with vector extensions. The figure contains 2 bars for each sparsity level: labeled Dedicated_HHT_1buffer and Dedicated_HHT_2buffer, representing the speedup achieved using

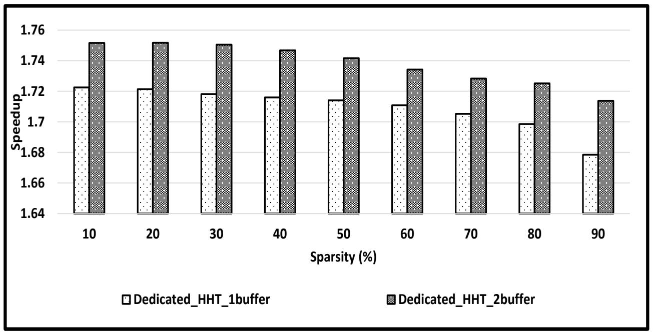

Fig. 4: *HHT* Speedup For *SpMV*

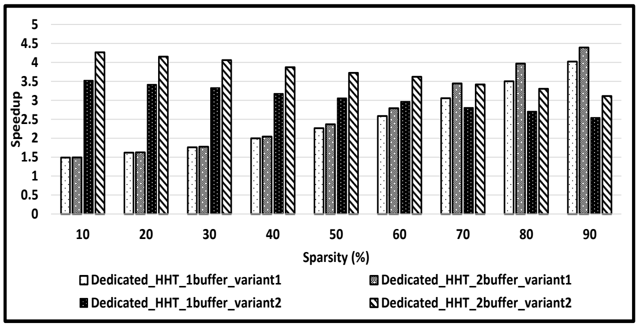

Fig. 5: *HHT* Speedup For *SpMSpV*

an ASIC *HHT* with 1 and 2 buffers. 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 figure, 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. 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 4) 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 5.2 and improves speedup. However, the minimal improvement over one buffer is because, the ASIC *HHT*is more than adequate to supply data to CPU and CPU will not be idling waiting for *HHT*.

HHT On SpMSpV Workloads: Figure 5 shows the performance gains achieved by *HHT* for sparse Matrix - sparse Vector (*SpMSpV*) multiplication. In Figure 4, we show the data for two variants of HHT for *SpMSpV*. In the first variant, *HHT* provides both matrix (row) values and vector values only if the nonzero values of matrix and vector are aligned. In the second variant, *HHT* only provides vector values corresponding to the location of nonzero matrix values: either a nonzero value if the corresponding vector location contains a value or zero otherwise. The figure contains four bars of data, two using a single buffer and two using two buffers for each of the two variants. The first two bars (Dedicated_HHT_1buffer_variant1 and Dedicated_HHT_2buffer_variant1) represent the speedups using an ASIC *HHT* for variant-1 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. As mentioned, in variant-1 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 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 5.2. In variant-2 *HHT* is 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, otherwise). The CPU is responsible for fetching non-zero values of the Matrix, and use the Vector values supplied by *HHT* to compute the results - we label these results as variant-2. In Figure 5, the next two bars (labeled Dedicated_HHT_1buffer_variant2 and Dedicated_HHT_2buffer_variant2) represent results where ASIC *HHT* only supplies Vector values (with one and two buffers). As can be observed, at lower sparsities, variant-2 *HHT* performs much better than baseline (CPU only) and better than variant-1 *HHT*. At higher sparsities, greater than 80% sparsities, variant-2 *HHT* performs worse than variant-1 *HHT* but better than baseline CPU version. This is because, at higher sparsities, the CPU is supplied with more zero values of Vector in variant-2 (when there is no matching Vector element corresponding to non-zero Matrix values) and these zero computations should be considered as wasted computations.[4] 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).

5.2. CPU - HHT Overlap

Our goal is to offload computations (or work) related to metadata processing (associated with locating 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. 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).

[4] CPU can ignore the zero multiplications to save energy -in the case of SIMD like vector registers, *HHT* can provide mask bits to skip unnecessary multiplications.

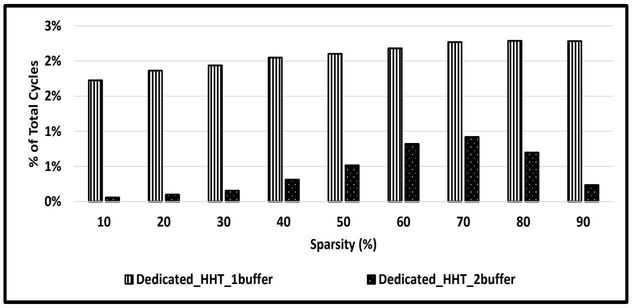

Fig. 6: CPU Wait Cycles For *SpMV*

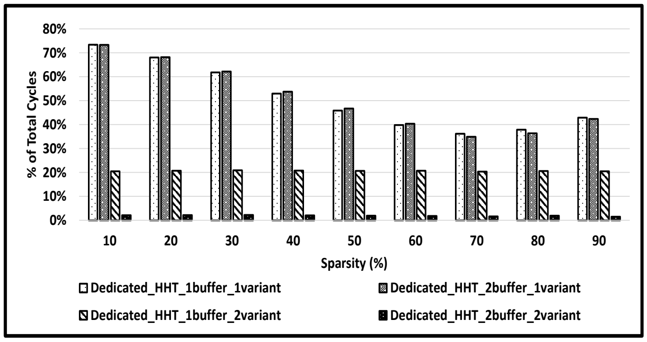

Fig. 7: CPU Wait Cycles For *SpMSpV*

Figure 6 shows the fraction of time CPU is idling (waiting for *HHT*) for sparse Matrix-dense Vector (*SpMV*) multiplication. The figure includes two bars for each sparsity level, and these bars parallel the bars shown in Figure 4. The two bars represent the fraction of the time CPU is idling when an ASIC *HHT* is used with one and two buffers. With an ASIC *HHT*, the application CPU rarely waits. This also results in maximum speedup as shown in Figure 4.

Figure 7 shows the CPU idle times for sparse Matrix - sparse Vector multiplication (*SpMSpV*). The four bars for each sparsity level parallel the four bars shown in Figure 5 (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). When *HHT* is supplying aligned Matrix and Vector non-zero values (variant-1), CPU is idling for a significant fraction of the total execution time. Two buffers show only minor improvements. When *HHT* only provides Vector elements (variant-2) the CPU idle times are significantly reduced.

5.3. 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.

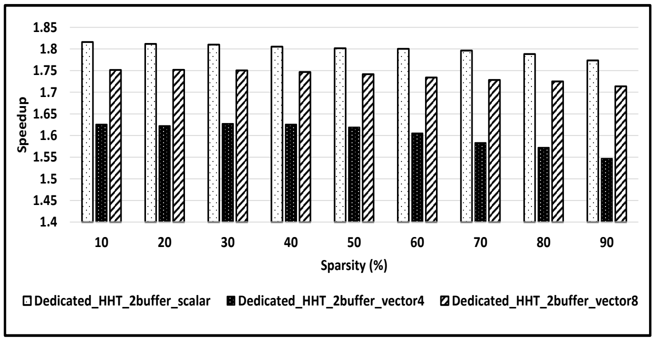

Fig. 8: *HHT* Sensitivity to Vector Width

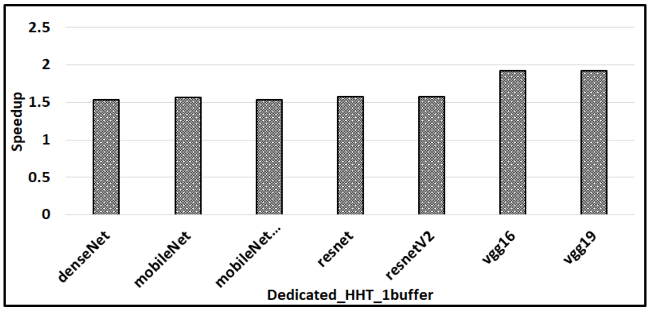

Fig. 9: *HHT* on DNN workloads

Figure 8 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 three bars for each sparsity (for a 512*512 sparse matrix) corresponding to three different vector widths, 1(scalar), 4, 8. 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), indicating that the ASIC *HHT* with double buffering can meet the demands of supplying needed values to CPU.

5.4. 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 [31], MobileNetV2 [32], DenseNet [33], ResNet [34], ResNetV2 [35], and VGG16, & VGG19 [36]. Figure 9 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 width vector width of 8. 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 [25].

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 is similar to the synthetic results at different sparsity and matrix sizes.

5.5. Area, Power and Energy Estimates

In this section, performance estimates of variant-2 *HHT* are presented by simulating System Verilog designs of *HHT* and Ibex [37] RISCV core [5] using Synopsys Design Compiler and PrimeTime tools. The area estimates of ASIC *HHT* is the sum of the logic gates of the control unit and storage for pipeline stages, two *HHT* memory side buffers of size 8, memory-mapped registers, internal state registers and one CPU side buffer. We synthesized three different feature sizes (28nm, 16nm and 7nm) using ARM libraries running at three different clock speeds (10 MHz, 50 MHz and 100 MHz). For this contribution, we present estimates of the design performing *SpMV* using 16 nm feature size running at 50 MHz clock for both RISCV and *HHT*. Our *HHT* is approximately 38.9% the size of an Ibex core.

Using the design described above for 16 nm at 50 MHz clock, we calculated energy savings using a 16*16 sparse matrix with 10% sparsity. RISCV core along with *HHT* requires 314 uW power to perform the index accessing overlapped with matrix-vector multiplication while the RISCV core alone requires 223 µW power. This is expected because RISCV plus HHT includes two separate processing elements. However, due to the compute-memory overlap of execution in RISCV with *HHT*, number of cycles required to complete the indexing and computation decrease compared to baseline RISCV alone. Thereby on average, the use of *HHT* results in 19% reduction in energy consumed for *SpMV* across different sparsities ranging from 10% to 90%, when compared with baseline. Any bigger matrices can be broken into 16*16 [6] sized matrices on *HHT* and supply vector values to RISCV core.

6. RELATED WORKS

Sparse Matrix Accelerators. Accelerating sparse matrix operations has received attention from both the hardware and software communities. On the hardware side, works propose hardware acceleration of the entire computation: some of these works include a CAM-based accelerator [38], accelerator for very large *SpMV* [39]. The work in [39] 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 memory bottleneck faced by vectorized codes running on traditional cores. Some researchers explored hardware that expands sparse data into dense by inserting zeroes [40], [23]. 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 [41] 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 [42] 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 [43] proposed a parallel sparse Matrix - sparse Vector (*SpMSpV*) algorithm that stores the product of sparse Matrix - dense Vector based on the row indices and later accumulates it, all by using buckets.
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 [44] focuses on support for flexible sparse matrix and vector multiplications. Sparse data is represented as bit-vectors and dataflow like Multiply-Accumulate units are configured based on the non-zero values in data. Moreau, et.al. [45] 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 [46,47]). 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 [48].
New Sparse Representations. In a different vein, there have been proposals on improving compression of sparse matrices and proposed techniques including hierarchical bit vectors [21] or compression on top of CSR [22]. There are proposals for specialized hardware to compress and decompress data for use by CPU (assuming that the CPU uses conventional *SpMV* software) [22]. Others propose hardware to use the new compression formats (such as hierarchical bit maps) for performing sparse matrix computations [21]. We programmed *HHT* to

[5] https://github.com/lowRISC/ibex

[6] Due to the limitations of the Synopsys tool available to us, we were unable to obtain the results for larger matrix size but we are trying to overcome this limitation.

handle sparse data represented using SMASH [21] 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 [21].

7. CONCLUSIONS

In this work, we presented a heterogeneous architecture consisting of a memory-side accelerator along with a general purpose processor 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 a dedicated or ASIC hardware for *HHT*. Our approach should be distinguished from other accelerators that accelerate the entire computation, not just index computations.

We evaluated the use of our *HHT* for sparse Matrix-dense Vector (*SpMV*), sparse Matrix - sparse Vector (*SpMSpV*) multiplications as well as convolution algorithms. The use of *HHT* offloads the index computations needed for sparse data representation to *HHT* and these computations are overlapped with the computations of the primary CPU core. 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. 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. We also show an average energy savings of 19% when ASIC *HHT* is used compared to baseline (for *SpMV*).

To provide flexibility of sparse data representations (e.g., CSR, COO, Bit vector, SMASH [21]), it may be worth considering a programmable *HHT*, using a simple RISCV like core. Such a *HHT* core can be even simpler than traditional 32-bit integer RISCV. It can be designed with very few integer instructions, very few integer registers, 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.

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. The authors acknowledge the valuable suggestions provided by SRC industry liaisons Mahesh Mehendale (TI), Chris Tsongas (TI), Jaekyu Lee (ARM) and Mihai Caraman (NXP).

8. REFERENCES

[1] NXP, "NXP microcontrollers overview," 2019.

[2] TI, "MSP432P401R, MSP432P401M simplelink mixed-signal microcontrollers," 2019.

[3] NXP, "Microprocesssors and microcontrollers," 2019.

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

[5] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel S. Emer, Stephen W. Keckler, and William J. Dally, "SCNN: an accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. 2017, pp. 27–40, ACM.

[6] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra, "Hello edge: Keyword spotting on microcontrollers," *CoRR*, vol. abs/1711.07128, 2017.

[7] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[8] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu, "A comprehensive survey on graph neural networks," *CoRR*, vol. abs/1901.00596, 2019.

[9] Ariful Azad, Georgios Pavlopoulos, Christos Ouzounis, Nikos Kyrpides, and Aydin Buluç, "Hipmcl: a high-performance parallel implementation of the markov clustering algorithm for large-scale networks," *Nucleic Acids Research*, vol. 46, pp. 1–11, 01 2018.

[10] Sandeep R. Agrawal, Christopher M. Dee, and Alvin R. Lebeck, "Exploiting accelerators for efficient high dimensional similarity search," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, 2016, pp. 3:1–3:12.

[11] Guoming He, Haijun Feng, Cuiping Li, and Hong Chen, "Parallel simrank computation on large graphs with iterative aggregation," in *Proceedings of the 16th ACM*

SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010, Bharat Rao, Balaji Krishnapuram, Andrew Tomkins, and Qiang Yang, Eds. 2010, pp. 543–552, ACM.

[12] Ariful Azad, Aydin Buluç, and John R. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015*. 2015, pp. 804–811, IEEE Computer Society.

[13] Aydin Buluç and John Gilbert, "The combinatorial blas: Design, implementation, and applications," *IJHPCA*, vol. 25, pp. 496–509, 12 2011.

[14] Felix Gremse, Andreas Höfter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann, "Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging," *SIAM Journal on Scientific Computing*, vol. 37, pp. C54–C71, 01 2015.

[15] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Phys. Rev. E*, vol. 76, pp. 036106, Sep 2007.

[16] Kjeld Schaumburg, Jerzy Wasniewski, and Zahari Zlatev, "The use of sparse matrix technique in the numerical integration of stiff systems of linear ordinary differential equations," *Computers & Chemistry*, vol. 4, no. 1, pp. 1–12, 1980.

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

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

[19] Aydin Buluç, Samuel Williams, Leonid Oliker, and James Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*. 2011, pp. 721–733, IEEE.

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

[21] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri-Ghiasi, Taha Shahroodi, Juan Gómez-Luna, and Onur Mutlu, "SMASH: co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. 2019, pp. 600–614, ACM.

[22] Arjun Rawal, Yuanwei Fang, and Andrew A. Chien, "Programmable acceleration for sparse matrices in a data-movement limited world," in *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. 2019, pp. 47–56, IEEE.

[23] Shashank Adavally, Nagendra Gulur, Krishna Kavi, Alex Weaver, Pranoy Dutta, and Benjamin Wang, "Express: Simultaneously achieving storage, execution and energy efficiencies in moderately sparse matrix computations," in *The International Symposium on Memory Systems*, 2020, pp. 46–60.

[24] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas, "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 178–190.

[25] Intel, "Overview: Intrinsics for intel® advanced vector extensions 2 (intel® avx2) instructions," 2019.

[26] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanaël Prémillieu, Alastair Reid, Alejandro Rico, and Paul Walker, "The ARM scalable vector extension," *CoRR*, vol. abs/1803.06185, 2018.

[27] RISCV Foundation, "The RISCV vector extension specification," 2020.

[28] Abraham Gonzalez, "The RISC-V ISA Simulator," 2019.

[29] RISCV Foundation, "RISC-V: The free and open RISC instruction set architecture," 2020.

[30] Timothy A. Davis and Yifan Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011.

[31] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017.

[32] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," *CoRR*, vol. abs/1801.04381, 2018.

[33] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger, "Densely connected convolutional networks," *CoRR*, vol. abs/1608.06993, 2016.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Identity mappings in deep residual networks," *CoRR*, vol. abs/1603.05027, 2016.

[36] Karen Simonyan and Andrew Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun, Eds., 2015.

[37] lowRISC, "Ibex: An embedded 32 bit risc-v cpu core," 2017.

[38] Leonid Yavits and Ran Ginosar, "Sparse matrix multiplication on CAM based accelerator," *CoRR*, vol. abs/1705.09937, 2017.

[39] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C. Hoe, Larry T. Pileggi, and Franz Franchetti, "Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. 2019, pp. 347–358, ACM.

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

[41] Aydin Buluç and John R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM J. Scientific Computing*, vol. 34, 2012.

[42] Joseph L. Greathouse and Mayank Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Piscataway, NJ, USA, 2014, SC '14, pp. 769–780, IEEE Press.

[43] Ariful Azad and Aydin Buluç, "A work-efficient parallel sparse matrix-sparse vector multiplication algorithm," in *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. 2017, pp. 688–697, IEEE Computer Society.

[44] E. Qin, A. Samajdar, H. Kwon, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," Feb 2020.

[45] T. Moreau, T. chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "A hardware-software blueprint for flexible deep learning specialization," *IEEE Micro*, Sept/Oct 2019.

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

[47] M. Rezaei and K. Kavi, "Intelligent memory manager: Reducing cache pollution due to memory management functions," *Elsevier Journal of Systems Architecture*, vol. 52, no. 2, pp. 207–219, Jan 2006.

[48] L.M. Fox, C.R. Hill, R.K. Cytron, and K.M. Kavi, "Optimization of storage-referencing gestures," in *Workshop on Compilers and Tools for Constrained Embedded Systems (CTES-2003), held in conjunction with Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES-2003)*, Oct. 29 2003.