Sparse Matrix-Vector Multiply on the HICAMP Architecture

John P. Stevenson
Stanford University
jpeter@stanford.edu

Amin Firoozshahian
HICAMP Systems
aminf13@hicampsystems.com

Alex Solomatnikov
HICAMP Systems
sols@hicampsystems.com

Mark Horowitz
Stanford University
horowitz@stanford.edu

David Cheriton
Stanford University
cheriton@stanford.edu

ABSTRACT

Sparse matrix-vector multiply (SpMV) is a critical task in the inner loop of modern iterative linear system solvers and exhibits very little data reuse. This low reuse means that its performance is bounded by main-memory bandwidth. Moreover, the random patterns of indirection make it difficult to achieve this bound. We present sparse matrix storage formats based on deduplicated memory. These formats reduce memory traffic during SpMV and thus show significantly improved performance bounds: 90x better in the best case. Additionally, we introduce a matrix format that inherently exploits any amount of matrix symmetry and is at the same time fully compatible with non-symmetric matrix code. Because of this, our method can concurrently operate on a symmetric matrix without complicated work partitioning schemes and without any thread synchronization or locking. This approach takes advantage of growing processor caches, but incurs an instruction count overhead. It is feasible to overcome this issue by using specialized hardware as shown by the recently proposed Hierarchical Immutable Content-Addressable Memory Processor, or HICAMP architecture.

Categories and Subject Descriptors

G.1.3 [**Numerical Linear Algebra**]: Sparse, structured, and very large systems—*direct and iterative methods*

Keywords

SpMV, Deduplication, HICAMP

1. INTRODUCTION

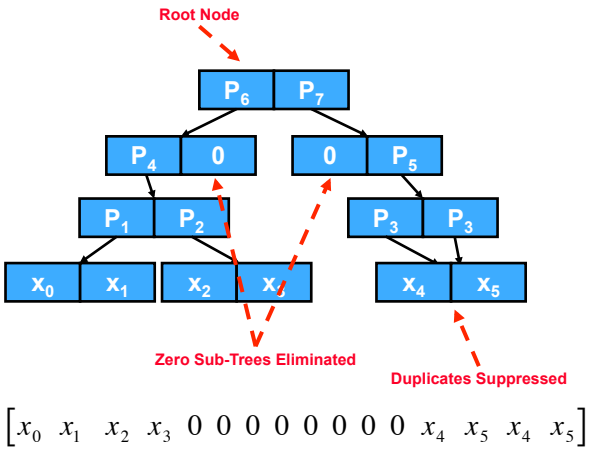
The ability to quickly invert a large sparse matrix is strongly desired by many computing applications. In practice, this reduces to finding the vector x which satisfies $y = Ax$, i.e., an explicit inverse is not computed. Direct approaches factor the matrix into upper and lower triangular components and then do forward and back-substitution. The cost of the direct approach can be amortized by solving several vectors against the same matrix. Unfortunately, finding the matrix factors can be quite costly, which motivates the use of iterative approaches. Iterative approaches propose a trial solution x_0, calculate the residual $r = y - Ax_0$, and update the trial solution based on the residual. The process repeats until the norm of the residual is brought to a satisfactorily low value. Computing the matrix-vector product on each pass comprises the bulk of the work in such a method. This fact has motivated much research into optimizing the performance of sparse matrix-vector multiply, or SpMV.

SpMV is typically limited by memory bandwidth because there is very little data reuse. If the total size of the working set exceeds the capacity of the last-level cache, the cache is effectively rendered useless: each matrix value must be brought in from DRAM, to be used exactly once, on each pass through the SpMV kernel. Much effort has focused on maximizing the attained bandwidth when using a matrix representation such as compressed sparse row (CSR) [3, 12, 19, 20, 22, 25], while relatively little work has focused on reducing memory traffic [2,13–15,24]. Using current techniques, SpMV execution time is set by the ratio of the working set size to the maximum sustainable memory bandwidth of a given system [23]. This motivates research into new methods that work around the bandwidth limitation.

This paper investigates one such method using matrix formats based on fine-grained memory deduplication. Our work is informed by the recently proposed Hierarchical Immutable Content-Addressable Memory Processor, or the HICAMP architecture [9]. The HICAMP architecture provides improved concurrent programming semantics and reduces the cost of inter-process communication by providing an interface to a main memory that is *snapshotted* or *versioned* on every update. To achieve this, without requiring an exorbitant amount of DRAM, HICAMP implements fine-grained in-memory deduplication. Such a deduplicated memory is useful in other domains as well: by introducing another level of indirection, it enables large pages *and* fine-grained sharing of page content in a virtualized server environment.

Such a memory suppresses duplicate zero values and thus inherently provides *sparsity oblivious* hardware support for compact representation of sparse matrices. Because some aspects of the HICAMP architecture may prove useful in fully realizing the potential of our method, we start by describing deduplicated memory and then briefly describe key hardware components used by HICAMP. We then describe several new matrix formats that rely on a deduplicated backing store and investigate their storage efficiency. One such format leverages the deduplicated memory to automatically

ICS'12, June 25–29, 2012, San Servolo Island, Venice, Italy.



$$v = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_4 & x_5 & x_4 & x_5 \end{bmatrix}$$

Figure 1: HICAMP DAG Storage: vector v is stored in the leaf nodes of the DAG

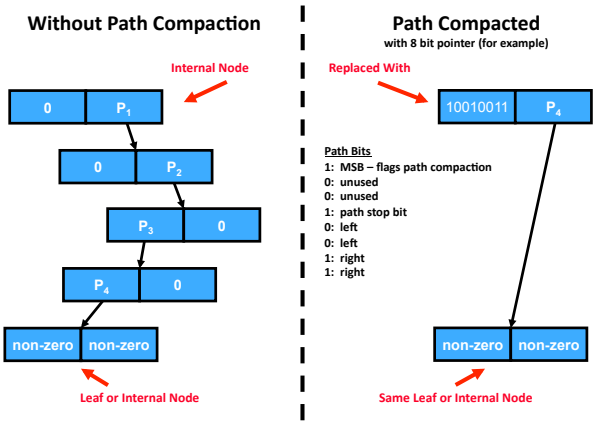


Figure 2: Path Compaction

detect symmetric matrix components and recursive patterns of content. Our best compaction result is an improvement of 5700x. These results encouraged us to implement a custom SpMV kernel that emulates the HICAMP data access pattern while running on a standard x86/Linux server. Using this kernel, we achieve an average speedup of 1.5x and a best-case speedup of 3.7x. We investigate the speedup and our performance limits using profiling techniques. Through this paper, we hope to provide insight to how a deduplicated memory could be used in conjunction with certain hardware components to further accelerate SpMV.

2. DEDUPLICATED MEMORY

The HICAMP architecture [8,9] provides the abstraction of a linear memory, but deduplicates memory content at a fixed granularity that we refer to as a *memory line*. To do this, a level of indirection is introduced. In particular, a given linear array is stored in the leaf nodes of a directed acyclic graph (DAG). The DAG itself is comprised of unique and immutable memory lines. To provide efficient addressing of the fixed granularity memory lines, we use line addresses that we individually refer to as a *physical line id* or PLID. To provide efficient reclamation, each PLID has an affiliated reference count. Figure 1 shows an example of some content (a sparse vector) mapped to a HICAMP DAG.

In Figure 1, the root node contains PLIDs 6 and 7. The memory line containing content (values) x_4 and x_5 is referred to by PLID 3 and has a reference count of two. The zero-valued PLIDs imply zero-valued subtrees. Although we illustrate here using an internal node fanout of two, this conceptual view of memory can use any branching factor. If all leaf nodes were unique and non-zero, the DAG shown in Figure 1 would be a fully balanced binary-tree. Using this observation, we adopt the following convenient (if informal) terminology for use in this paper: when necessary to specify the branching factor, we describe a DAG such as that illustrated in Figure 1 as a binary-tree (or *b-tree*). We describe a DAG with a branching factor of four as a quad-tree (*q-tree*) and a DAG with a branching factor of eight as an oct-tree (*o-tree*).

Because sparse data is common, not just in linear algebra, an optimization is used when internal DAG nodes become underutilized. A path-compacted node replaces a sequence of nodes containing only one PLID each by pointing directly to the PLID at the end and by encoding the path through the DAG. A path compaction is flagged by a single bit sacrificed from the pointer space (example shown in Figure 2).

2.1 HICAMP Architecture

The HICAMP architecture provides hardware support for efficient concurrency-safe access to shared data. Most research has focused on the model of multiple threads accessing data within a single shared address space, but many real applications use multiple separate processes with separate address spaces for fault-isolation and simplified synchronization. HICAMP bridges this discrepancy and provides process-like shared data protection through snapshot isolation and reduces the high cost of inter-process communication by allowing pointer sharing.

HICAMP translates load and store instructions into operations on DAGs of unique and immutable memory lines. The memory lines in the DAG, both internal and leaf, are hardware protected and cannot be modified directly by software. Instead, HICAMP provides a *virtual segment table* that maps from software visible *segment ids* to the DAG root nodes. The segment table provides the interface to software and is somewhat analogous to a TLB.

When a thread begins to access memory referred to by a segment id, it issues a command to hardware that accesses the segment table and issues a reference count increment to the DAG root node. This reference count increment guarantees that the root node, and by extension, any PLID or leaf node referred to by the root node, will not be deallocated or modified. Thus, the thread has a stable and isolated snapshot of the memory content. The thread can modify its view of the DAG, but changes are not globally visible until committed by an update to the virtual segment table. Read-write conflicts are thus eliminated and write-write conflicts can be detected by compare-and-swap (CAS) against the previous root node PLID. This enables snapshot and transactional semantics for arbitrarily large regions of memory, i.e. independent of cache size.

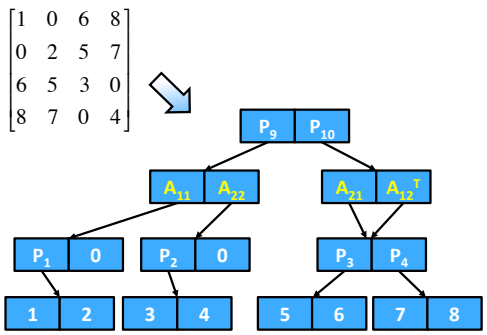


Figure 3: A Sparse Symmetric Matrix Mapped to QTS Format

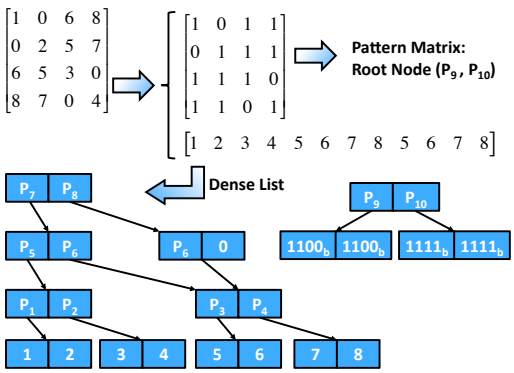


Figure 4: NZD Format using binary-tree and 8 bit nodes

Although we cannot here fully describe all aspects of the architecture, two specific hardware structures are relevant to this paper: the HICAMP memory controller (provides efficient deduplication of content) and the HICAMP iterator register (provides efficient access to DAG leaf nodes).

2.2 HICAMP Memory Controller

The HICAMP memory controller provides an interface to a content addressable memory that is backed by standard DRAM. To implement content lookup, the HICAMP memory controller breaks the address space into hash buckets and performs an efficient hash-based search. The HICAMP memory controller provides two fundamental operations: *lookup by content* (returns a PLID) and *read by PLID* (returns content). The PLID is comprised of the hash bucket index and way number and can be directly converted to a physical address. The HICAMP memory controller also maintains the reference count for each line.

2.3 HICAMP Iterator Register

The HICAMP iterator register accepts load and store instructions from software as operations relative to some *segment id*. A load operation accepts an offset and traverses from the graph root (found in the *virtual segment table*) to the leaf specified by the offset. A store operation implies the insertion (or replacement) of some content as a leaf node. The iterator register issues a lookup to the memory controller and then inserts the returned PLID as an internal node in the DAG. This procedure of lookup-by-content and node insertion continues until the DAG root node is replaced with a new PLID.

As the name implies, the iterator register is meant to provide efficient iteration over leaf elements that are logically contiguous, but actually scattered throughout memory. Besides implementing DAG traversal in hardware, the iterator register maintains a private cache of DAG elements along the path from the root node to the currently referenced leaf node to minimize memory accesses.

3. DEDUPLICATED MATRIX FORMATS

Using the memory structures described in Section 2, a sparse matrix can be efficiently stored with implicit indices (i.e., as if it were dense) because the DAG eliminates zero valued sub-graphs. We call this a logically dense format. A common approach to sparse matrix storage is to store a list of non-zero values along with arrays that indicate their indices. We call this a compressed format when mapped to a deduplicated memory. A hybrid of these two can be employed – this approach is useful when a large amount of non-zero pattern symmetry exists, but the non-zero values themselves are unique. In this section, we describe the deduplicated matrix formats that we studied. We begin with the logically dense formats (RMA & QTS), then move to hybrid (NZD), and then to compressed (HCSR & HCOO).

3.1 Row Major Array (RMA)

The most basic HICAMP matrix format is row major array. This format logically stores *every* matrix value in row major order; i.e., it is logically dense. Although lacking in sophistication, this method achieves storage efficiency at parity with the canonical CSR format (see Table 2).

3.2 Quad-Tree Symmetric (QTS)

Another logically dense format, quad-tree symmetric, divides the matrix into four quadrants (A_{11}, A_{12}, A_{21}, and A_{22}) and stores each in a separate sub-graph. When the sub-matrix is on diagonal, the storage order is A_{11}, A_{22}, A_{21}, then A_{12}^T. In this way, the QTS format automatically exploits the duplicate content found in symmetric matrices. Moreover, because it makes no special provision for explicitly symmetric matrices, it automatically mines out *any amount* of symmetry, i.e., even for non-symmetric matrices. In Figure 3, we show an example mapping a small symmetric matrix to a QTS encoded DAG.

3.3 Non-Zeros Dense (NZD)

The non-zeros dense format is a hybrid format that uses two DAGs: one representing a sparse pattern matrix and the other a dense vector. In NZD, the underlying non-zero pattern is factored out as a pattern matrix in QTS form. Because the pattern matrix leaf elements are logically only one bit in size, 64 leaves can fit in the same space as one double precision floating point value. The non-zero values are then stored in a dense list ordered to correspond to the QTS pattern matrix. In Figure 4, we show the NZD format using the toy matrix from Figure 3.

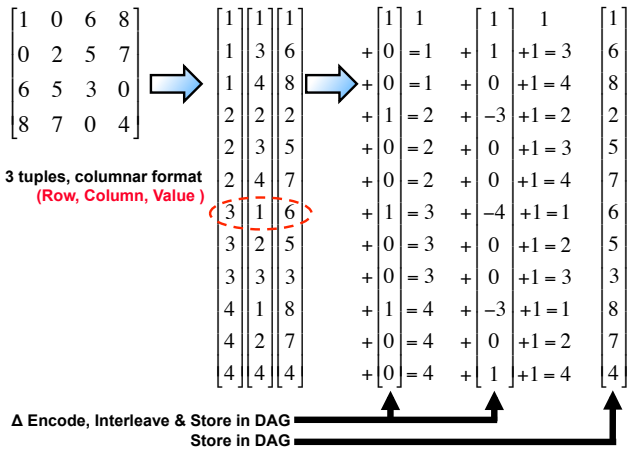


Figure 5: HCOO Format: Delta Encoding and Interleaving Scheme

3.4 HICAMP Compressed Sparse Row (HCSR)

The HICAMP compressed sparse row format maps a list of non-zero values, a corresponding list of column indices, and a shorter list that indicates the positions of row breaks in both of the former lists to the leaf nodes of three separate DAGs. As the name implies, this is a mapping of the canonical compressed sparse row (CSR) format into a deduplicated memory.

To enhance deduplication, the column indices and row breaks are delta encoded. Delta encoding exposes more duplicate values where regular patterns exist. For example, the sequence $1, 2, 3, 4, 10$ becomes $1, 1, 1, 1, 6$ when delta encoded (note that the first value in the list is the seed value). The same sequence can be delta encoded as $1, 0, 0, 0, 5$ with an assumed increment of one. Because the column indices are often clustered in sequential order and because our method favors zero values, we use an assumed increment of one when delta encoding the column indices list.

3.5 HICAMP Coordinate Format (HCOO)

In the traditional coordinate format (COO), each non-zero value is stored along with explicit matrix indices, i.e., as a 3-tuple: (row index, column index, non-zero value). In the HCOO format, we first arrange the tuples in columns so that we have a list of non-zero values, a list of row indices, and a list of column indices. To enhance deduplication, we apply delta encoding to the index lists (with an assumed increment of one for column indices). Although the index lists are logically the same length as the list of non-zero values, the index list data elements are one half the width of the double precision floating point values. Thus, we next interleave the row and column indices so that we have only two lists, each with the same number of bits. We then map these two lists to the leaf nodes of two separate DAGs. We show this delta encoding and interleaving scheme in Figure 5. The two DAGs, thus obtained, are logically the same height – this, by itself, is of no particular importance, but in our software implementation it allows us to track the state of only one DAG and thus reduce our overall instruction count.

3.6 Storage Bounds

Our deduplicated storage formats can achieve a high degree of data compaction, but also provide reasonable bounds in the worst case. Given an m by n matrix A, with nnz random non-zero values, and pessimistically assuming that no deduplication occurs for the row and column indices, we can precisely compute the DAG overhead for the compressed formats HCOO and HCSR using a geometric series:

$$total\_nodes = \begin{cases} 2 \cdot num\_leaf\_nodes & b\text{–}tree \\ 4/3 \cdot num\_leaf\_nodes & q\text{–}tree \\ 8/7 \cdot num\_leaf\_nodes & o\text{–}tree \end{cases} \quad (1)$$

For HCOO and HCSR the number of leaf nodes is $O(nnz)$ and thus the storage is no worse than $O(nnz)$. For logically dense formats, we assume that each non-zero costs one leaf node and that each leaf costs $log(m \cdot n)$ total nodes. With the understanding that n represents the longer dimension of the matrix, the storage cost for a logically dense format is no worse than $O(nnz \cdot log(n))$.

4. CONCURRENT SYMMETRIC SPMV

Using matrix symmetry, traditional matrix storage formats and SpMV kernels can save on storage and bandwidth requirements by storing only the upper (or lower) half of the matrix. The symmetric-CSR format keeps only the upper half and thus achieves an immediate ∼2x storage advantage.

In concurrent SpMV, threads are normally assigned distinct matrix rows. This works well for non-symmetric CSR SpMV because, for each thread, it preserves the property of sequential access along a matrix row. More importantly, this work partition assigns a disjoint write set to each thread, and thus the threads can operate in complete independence with no synchronization.

Unfortunately, this partitioning inhibits performance for concurrent symmetric SpMV: because each thread now owns both one row and one column of the matrix, each thread can write to any value in the output vector. Thus, atomic updates are required for every write operation and synchronization overhead eliminates the benefit of parallelism.

The QTS format described in Section 3.2 neatly avoids these issues. Using QTS, any work partition that is thread-safe for a non-symmetric matrix is inherently thread-safe for a symmetric matrix. To avoid the need for thread synchronization, we partition the QTS matrix by row. In Section 6.9, we demonstrate that our symmetric concurrent SpMV kernel scales with thread-count and exploits the reduction in memory traffic that matrix symmetry enables.

5. SOFTWARE IMPLEMENTATION

5.1 Matrix Encoding

We implemented all of the matrix formats described in Section 3 using software to perform the deduplication. For each format, we implemented binary, quad, and oct-tree variants. For binary internal tree-degree, we used 8-byte memory lines, i.e., wide enough to store one double precision floating point value as a leaf node and enough for two 32-bit pointers in internal nodes. For quad-tree, we used 16-byte memory lines (two doubles per leaf node and four 32-bit pointers per internal node), and for oct-tree, 32-byte memory lines.

```
procedure TraverseDAG( nodes, height ):
  sp = 1                # initialize stack pointer
  ptrStack[0] = 0       # node pointer stack
  offStack[0] = 0       # leaf node offset stack
  lvlStack[0] = height  # DAG level stack
  while( sp ):
    sp -= 1
    level  = lvlStack[ sp ]
    offset = offStack[ sp ]
    ptr    = ptrStack[ sp ]
    node = nodes[ ptr ]
    if level == 0:
      print "offset =", offset, ", leaf =", node
    else:
      if node.left & pathbit:
        # decode path & push new offset, level, & node pointer
      else:
        if node.right:
          ptrStack[ sp ] = node.right
          lvlStack[ sp ] = level
          offStack[ sp ] = offset + 1 << level
          sp += 1
        if node.left:
          ptrStack[ sp ] = node.left
          lvlStack[ sp ] = level
          offStack[ sp ] = offset
          sp += 1
```

Figure 6: DAG Traversal Code

5.2 SpMV Kernel

We implemented an SpMV kernel for each of the HICAMP formats. Due to space limitations, we present results only for HCOO and QTS because they are the most instructive. For SpMV, we used DAGs with internal node fanout of four (quad-tree) because this degree achieves higher data-compaction than oct-tree while exposing more memory parallelism than binary-tree. In Figure 6, we show pseudo-code for traversing a DAG. For simplicity, we show code for binary-tree. This generic code tracks two explicit state variables: the current DAG level (level zero indicates a leaf node) and the offset from the logically leftmost leaf node. In our QTS kernel, we replace the offset state variable with row/column state variables. In the HCOO SpMV kernel, we do not track the offset because the row/column indices are given explicitly in the leaf node data: DAG traversal stops when a zero-valued left subtree is encountered. Further, because the HCOO format stores dense lists, the HCOO kernel does not check for path compactions. Although the HCOO kernel must logically traverse two DAGs, as mentioned in Section 3.5, these DAGs have exactly the same number of leaf nodes and can thus share their state.

6. PERFORMANCE RESULTS

6.1 Methodology

We report performance results for static memory requirements (memory footprint) and for SpMV execution time speedup. In addition to reporting raw performance, we measure memory traffic, instructions executed, and instructions per cycle (IPC). We compare our HCOO SpMV kernel to a multi-threaded CSR SpMV and we compare our QTS SpMV kernel to a symmetry aware CSR SpMV kernel. We measure execution time at nano-second resolution using calls to `clock_gettime()` and average the results over many trials. After the timing run, we measure instruction and cycle count using the Linux `perf` tool and we measure DRAM load and store operations using LIKWID [21] to access uncore performance counters.

Sockets	2
Cores	12
Threads	24
LLC	12 MB
f_{clk}	2.93 GHz
Mem Channels	3
f_{mem}	1066 MHZ
Mem BW	25.6 GB/sec

Table 1: Test Platform Specifications

6.2 Test Matrices & Performance Metrics

A set of 74 non-symmetric matrices and 12 symmetric matrices were drawn from the UF Sparse Matrix Repository [10]. We ensure that the underlying data span a number of disciplines. For an m by n matrix with nnz non-zero elements, we calculate its non-deduplicated memory footprint as follows:

$$bytes_{csr} = 12 \cdot nnz + 4 \cdot m + 4 \tag{2}$$

$$working\_set\_size_{csr} = bytes_{csr} + 8 \cdot m + 8 \cdot n \tag{3}$$

And the deduplicated memory footprint is as follows:

$$bytes_{dag} = bytes\_per\_node \cdot num_{nodes} \tag{4}$$

$$bytes\_per\_node = \begin{cases} 8 & b\text{–}tree \\ 16 & q\text{–}tree \\ 32 & o\text{–}tree \end{cases} \tag{5}$$

$$working\_set\_size_{dag} = bytes_{dag} + 8 \cdot m + 8 \cdot n \tag{6}$$

We compute our key performance metrics as follows:

$$matrix\_compaction = \frac{bytes_{csr}}{bytes_{dag}} \tag{7}$$

$$working\_set\_compaction = \frac{working\_set\_size_{csr}}{working\_set\_size_{dag}} \tag{8}$$

$$speedup = \frac{t_{spmv\_csr}}{t_{spmv\_dag}} \tag{9}$$

$$mem\_traffic = 64 \cdot (num_{dram\_rds} + num_{dram\_wrs}) \tag{10}$$

$$mem\_traffic\_advantage = \frac{mem\_traffic_{csr}}{mem\_traffic_{dag}} \tag{11}$$

$$mem_{BW} = \frac{mem\_traffic}{t_{spmv}} \tag{12}$$

6.3 Compaction Results

Table 2 shows compaction results for non-symmetric matrices for each HICAMP format using binary, quad, and oct-tree. In Figure 7, we plot the best achieved compaction ratio (Equation 7) versus non-deduplicated size (Equation 2). The results in Figure 7 are dominated by the QTS and NZD formats as these two methods simultaneously expose fine and coarse-grained patterns in the underlying matrices and take advantage of what symmetry does exist.

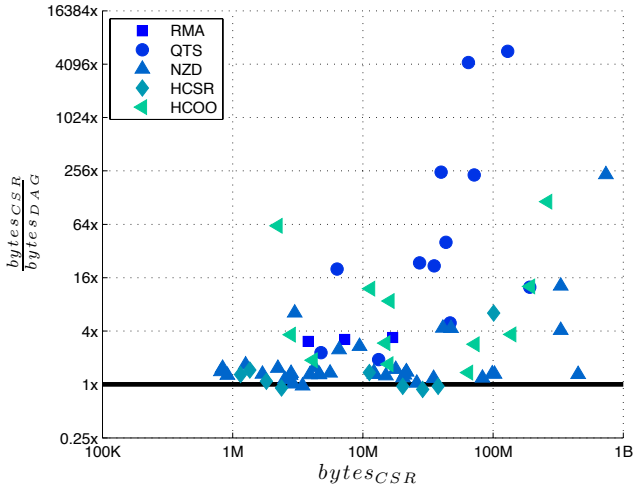


Figure 7: Matrix Data Compaction (Equation 7) vs. Uncompacted Size (Equation 2)

Format	Degree	Geomean	Best	Worst
RMA	b-tree	1.1x	18x	0.45x
	q-tree	0.8x	11x	0.28x
	o-tree	0.6x	8x	0.16x
QTS	b-tree	2.5x	5719x	0.46x
	q-tree	2.3x	5244x	0.29x
	o-tree	1.8x	4028x	0.18x
NZD	b-tree	2.5x	1182x	0.60x
	q-tree	2.8x	1160x	0.79x
	o-tree	2.6x	918x	0.79x
HCSR	b-tree	1.9x	1289x	0.54x
	q-tree	2.2x	1289x	0.76x
	o-tree	2.2x	979x	0.87x
HCOO	b-tree	1.8x	1513x	0.54x
	q-tree	2.2x	3185x	0.65x
	o-tree	2.0x	1141x	0.67x

Table 2: Comparison of HICAMP Matrix Formats

Matrix	Matrix Size	Uncompacted Working Set Size	Matrix Compaction	Working Set Compaction	Speedup
barrier2-1	25 MB	27 MB	0.9x	0.9x	0.9x
poisson3Db	28 MB	29 MB	0.6x	0.7x	0.4x
mc2depi	26 MB	34 MB	1.9x	1.6x	1.6x
TSOPF_RS_b300_c2	34 MB	34 MB	13.4x	11.6x	1.9x
thermomech_dK	33 MB	36 MB	0.9x	0.9x	0.7x
sme3Dc	36 MB	37 MB	0.9x	0.9x	0.7x
stat96v3	38 MB	47 MB	12.5x	4.0x	1.7x
xenon2	45 MB	47 MB	4.4x	3.7x	1.5x
webbase-1M	39 MB	55 MB	1.7x	1.4x	0.9x
rajat29	45 MB	55 MB	1.6x	1.4x	1.1x
stormG2_1000	42 MB	56 MB	6.5x	2.7x	1.6x
Chebyshev4	62 MB	63 MB	1.3x	1.3x	1.1x
largebasis	62 MB	68 MB	3.3x	2.7x	2.3x
pre2	69 MB	79 MB	2.8x	2.3x	1.3x
ohne2	79 MB	82 MB	1.0x	1.0x	0.9x
Hamrle3	69 MB	91 MB	85.9x	4.0x	2.2x
PR02R	94 MB	97 MB	1.1x	1.1x	1.1x
torso1	98 MB	100 MB	1.1x	1.1x	1.2x
Rucci1	97 MB	113 MB	5.7x	3.4x	2.8x
tp-6	133 MB	141 MB	2.3x	2.2x	1.5x
atmosmodl	124 MB	147 MB	3185.0x	6.4x	3.7x
TSOPF_RS_b2383	185 MB	186 MB	12.8x	12.3x	1.8x
circuit5M_dc	184 MB	237 MB	1.7x	1.5x	1.2x
rajat31	250 MB	322 MB	115.1x	4.4x	2.2x
cage14	316 MB	339 MB	2.0x	1.9x	1.0x
FullChip	316 MB	362 MB	1.7x	1.5x	0.9x
RM07R	430 MB	436 MB	1.2x	1.2x	1.1x
circuit5M	702 MB	787 MB	3.0x	2.5x	1.8x
average	n/a	n/a	124.0x	2.5x	1.5x

Table 3: HCOO Compaction and SpMV Speedup

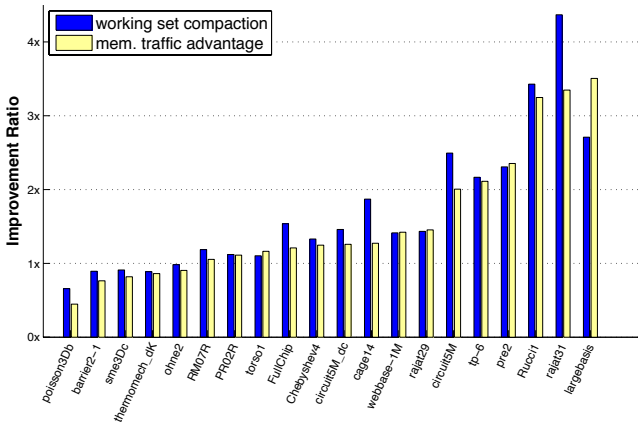


Figure 8: Memory Traffic Advantage and Matrix Compaction for Non-Cacheable Matrices

6.4 SpMV Execution Time

For SpMV performance evaluation, we restrict our attention to the 28 matrices whose working set size exceeds the capacity of the last level cache. Table 3 shows the speedup obtained for these matrices. We achieve an average speedup of 1.5x and a best-case speedup of 3.7x (for matrix *atmosmodl*).

Because our speedup lags the potential improvement indicated by the compaction results, in the remainder of this section, we investigate this discrepancy by analyzing memory traffic, memory bandwidth, instruction count, and IPC.

6.5 Memory Traffic

In Figure 8, we compare memory traffic advantage (Equation 11) to working set compaction (Equation 8). We observe that the reduction in memory traffic is indeed correlated to this compaction metric. As the amount of compaction increases, the probability of evicting a highly deduplicated node from the cache increases. This results in a diminishing return in terms of memory traffic as compared to working set compaction.

In Figure 8, we omit eight matrices whose working sets were compacted to a size smaller than the last level cache capacity. Although we would like to show these on the plot, their memory traffic ratio is heavily skewed in favor of SpMV speedup and the remaining data points are obscured. As a specific example, matrix *stat*96*v*3 had its working set compacted from 47 MB to 12 MB resulting in a memory traffic advantage of 90x. For *stat*96*v*3, the SpMV execution time speedup was only 1.7x. Obviously, the compaction technique is providing a substantial advantage in terms of memory traffic and one can conclude that DRAM bandwidth is not the performance limiting factor. In the following sections, we investigate performance limits by analyzing sustained bandwidth, instruction count, and IPC.

6.6 Memory Bandwidth

In Figure 9, we plot the sustained memory bandwidth for all matrices in our test set. Additionally, we show the peak theoretical bandwidth of 25.6 GB/sec and the maximum sustainable bandwidth obtained by running the synthetic benchmark *stream* [18]. As expected, the CSR implementa-

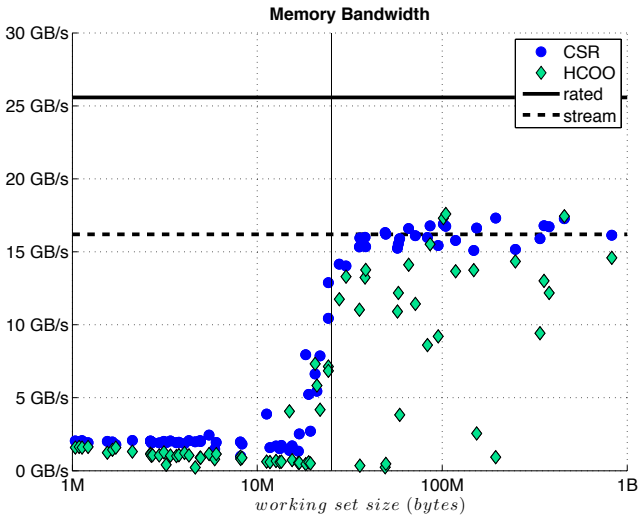


Figure 9: Sustained Bandwidth to DRAM

```
procedure RMA_SpMV( mtxSegID, m, n, y, x ):
  it = IterReg( mtxSegID )
  it.SetSkipZeroValuesOnIncrement()
  while( it.Pos() != it.End() ):
    v = it.Val()
    p = it.Pos()
    r = p / n
    c = p % n
    y[r] += v * x[c]
    it.PosInc()
```

Figure 10: SpMV Using an Iterator Register

tion of SpMV attains the maximum sustainable bandwidth to DRAM (as given by *stream*) when the working set size exceeds twice the capacity of the last level cache (solid vertical line in Figure 9).

6.7 Instruction Count

Although effective at reducing memory traffic, our method imposes the cost of significantly increased instruction count. We find that the CSR implementation executes approximately 9 instructions per non-zero value while our HCOO kernel requires, on average, 26. A significant amount of implementation effort was spent on minimizing the amount of generated code in the performance critical kernel, and despite this, the ratio is still quite unfavorable.

To illustrate that this code overhead would be significantly reduced by implementing hardware such as the HICAMP iterator register, we show pseudo-code for SpMV using the RMA format in Figure 10. Visually comparing to the code in Figure 6, it is clear that there are fewer lines of *source code*. More importantly, because the iterator register implements DAG traversal in hardware, the amount of *generated code* will be far less. In particular, a command such as `it.PosInc()` is reduced to a single hardware instruction. Moreover, the iterator register hardware provisions fast memory for the DAG nodes along the path to the current leaf node and concurrently maintains meta-data such as its offset from the leftmost leaf; the software implementation (Figure 6), is required to maintain a pointer stack, offset stack, and level stack. All of these benefits are retained

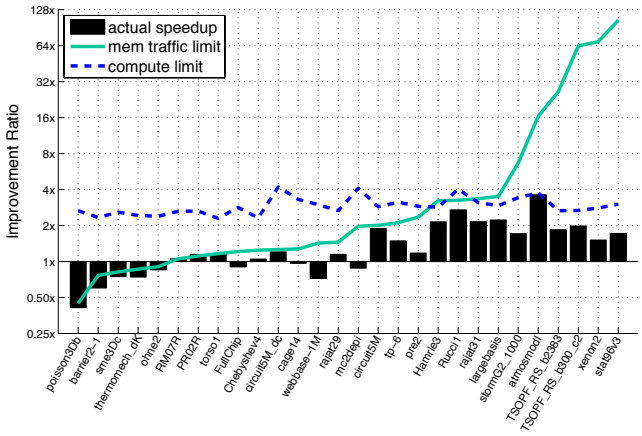


Figure 11: Performance Limits and Speedup

when extending the code shown in Figure 10 to handle other formats such as QTS, NZD, and HCOO.

6.8 Performance Limits

We conclude that the two relevant performance limits are main memory bandwidth and compute:

$$bw\_limit = \frac{mem\_traffic_{csr}}{mem\_traffic_{dag}} \quad (13)$$

$$compute\_limit = \frac{num_{instructions\_csr}/IPC_{csr}}{num_{instructions\_dag}/IPC_{dag}} \quad (14)$$

In Figure 11, we plot the obtained SpMV speedup (Equation 9) and compare it to the memory bandwidth limit (Equation 13) and the compute limit (Equation 14).

The bandwidth limit is plotted under the assumption that our kernel can achieve the maximum sustainable bandwidth as indicated by *stream*. This assumption is justified by the high sustained bandwidth exhibited by the data sets that do not achieve a significant reduction in working set size (see Figure 9). The compute limit is plotted under the assumption that our method can sustain an IPC of 2 (i.e. if not bandwidth limited) and that the CSR method achieves only an IPC of 0.5 (i.e. because it is always bandwidth limited). The average IPC numbers, and other data such as instruction count and DRAM operations, are based on profiling (described in Section 6.1) for the matrices shown in Figure 11.

Prior to the advent of multi-core, SpMV optimization research focused on adding work to the inner loop [22] so that maximum sustainable bandwidth could be attained. Our results shown in Figure 9 indicate that we achieve maximum sustainable bandwidth, but through the use of threads rather than optimizations that generate more memory accesses within a given code block. This is a result of current technology trends: the available memory bandwidth per core is dropping [1]. These technology trends and our results, based on measurements from actual hardware, imply that a HICAMP system provisioned with the same amount of cache as our test platform (Figure 1) could operate at the bandwidth limit and achieve a best-case speedup of 90x. In Figure 12, we simulate further decreases in available bandwidth per core by re-running our experiments using three,

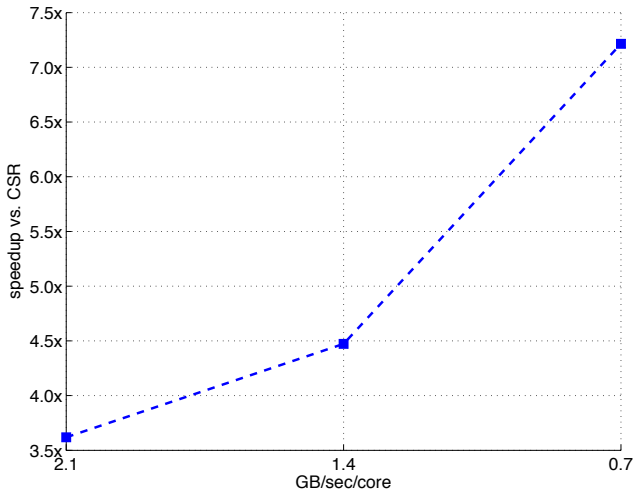


Figure 12: HCOO SpMV speedup vs. multi-threaded CSR as GB/sec/core drops (matrix *atmosmodl*)

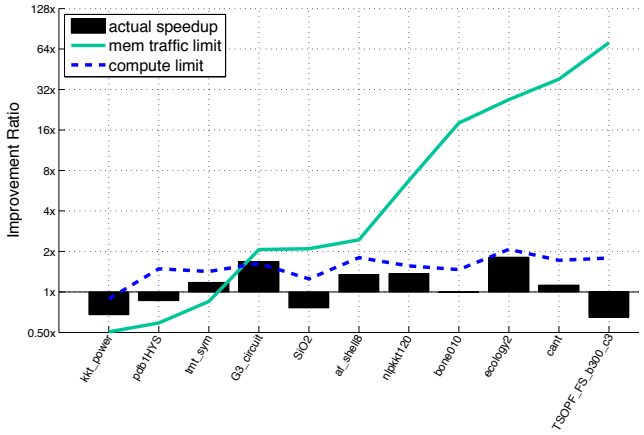


Figure 13: Performance Limits for QTS SpMV on Symmetric Matrices

then two, and then one memory channel: as the cores become more bandwidth starved, our method starts to significantly outperform the traditional CSR method.

6.9 QTS Symmetry Oblivious SpMV

We now assess the performance of our *symmetry oblivious* QTS SpMV kernel. We compare performance to single threaded symmetric CSR SpMV because this kernel achieves the lowest memory traffic. In Figure 13, we show the performance limits versus the speedup achieved. The best-case memory traffic advantage of 71x would be 142x (matrix *TSOPF_FS_b300_c3*) if we compared to a parallelized SpMV kernel without symmetric storage. In some cases, we transfer more data than required by symmetric CSR SpMV due to a combination of cache evictions and less raw compaction.

The QTS SpMV kernel has even more code overhead than our HCOO SpMV kernel with an average of 121 instructions per non-zero matrix element (compared to an average of 26 for HCOO). Because of this, its compute limit is lower and its raw performance lags. In Figure 14, we show that, when compute limited, our method scales very close to linearly with additional cores (matrix *nlpkkt120*). For many matrices, we have significant headroom to allow this scaling to continue as indicated by Figures 11 and 13.

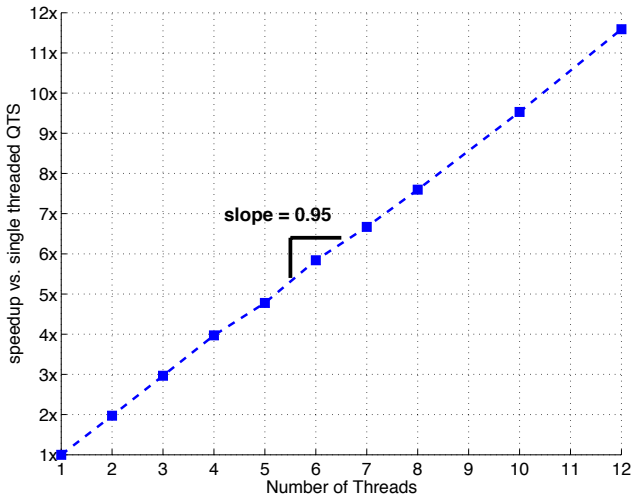


Figure 14: QTS SpMV Scaling With Threads (matrix *nlpkkt120*)

7. RELATED WORK

The literature on optimizing SpMV is vast: a testament to the difficulty and importance of this task. Historically, it was difficult to achieve peak performance due to the very tight inner loop of SpMV coupled with the indirect and random access pattern into the source vector. This motivated research into methods for adding extra work into the inner loop and regularizing data access patterns. Regularized data access can be achieved by finding dense matrix sub-blocks or through domain specific knowledge that allows a highly customized SpMV kernel to be invoked. Because the bandwidth limitation has long been recognized, research has also focused on methods for reducing memory traffic. Thus, most research into optimizing SpMV can be categorized as either reducing memory traffic, adding work to the inner loop, regularizing data access patterns, or some combination of these.

7.1 Non-Symmetric SpMV

The block compressed sparse row (BCSR) format captures all three of these tactics by using a CSR index structure to point to dense submatrices (or blocks). Because the blocks are fixed size and dense, they exhibit regular data access patterns and allow for optimal loop unrolling. Further, some matrices require less storage because of reduced index data overhead. Vuduc [22] and Williams [25], et al., describe *OSKI*, a framework for autotuning BCSR SpMV kernels.

With the advent of programmable GPUs, researchers have attempted to utilize the high GPU to video-memory bandwidth. Bell and Garland [3] describe methods to approach maximum sustainable bandwidth for SpMV on GPU through several techniques directed at regularizing data access. Because power constraints now force GPUs to include large on-die caches, the regular control flow of our HCOO format makes it an interesting candidate for GPU implementation.

Method	Best Compaction	Best Speedup	Index Compaction	Value Compaction	Data Reuse	Kernel Code
DCSR	1.4x	1.5x	Yes	No	No	Generic
PBR	1.5x	1.5x	Yes	No	Via Code[1]	Per Matrix
RPCSR	1.5x	1.5x	Yes	Yes	Via Code[1]	Per Matrix
CSR-DU	1.3x	1.8x	Yes	No	No	Generic
CSX		1.9x	Yes	No	Via Code[1]	Per Matrix
RSB		2.0x	Yes	No	No	Generic
CSR-VI	2.4x	2.5x	Yes	Yes	Yes	Generic
RBCSB		3.5x	Yes	No	No	Generic
DAG (this work)	5700x	3.7x	Yes	Yes	Yes	Generic
(1) Matrix structure embedded in custom generated code						

Table 4: Comparison of Non-Symmetric SpMV Compaction Techniques

To improve the concurrent scalability of computing $A^T x$, Buluç et al. introduced the new compressed sparse block (CSB) format [6] which has storage and performance similar to CSR. In the reduced bandwidth compressed sparse block (RBCSB) format [7], Buluç et al. use bitmasked register blocks (similar to the NZD pattern matrices in Section 3.3) to reduce the storage requirements of CSB.

Kourtis et al. [13,14], explore the effect of delta-encoding the index values (CSR-DU) and then add value deduplication via an indirection map (CSR-VI). Willcock and Lumsdaine [24] present the delta-coded sparse row (DCSR) method (similar to CSR-DU).

More sophisticated attempts to reduce memory traffic have used offline analysis followed by code generation (resulting in a custom SpMV kernel on a per matrix basis). In the compressed sparse extended (CSX) method [15], Kourtis analyzes each matrix separately to discover the underlying nonzero structure and customizes the delta encoding scheme for the index values. In their row pattern CSR (RPCSR) format [24], Willcock and Lumsdaine exploit macro-scale patterns in each matrix row. Finally, Belgin et al., observe that the underlying physical problem often generates a set of several distinct sub-matrix patterns. They exploit this in their pattern based-representation (PBR) [2].

Recursive matrix layouts, similar to our QTS format, have inspired recent papers [4,17]. In their recursive sparse blocks (RSB) format [17], Martone et al., use a quad-tree with Z-Morton ordering to store pointers to sparse submatrices that are sized to balance the work partitions. Because the quad-tree structure implies an offset (similar to our implicit indices), they use 16 bit index elements in the leaf submatrices.

In Table 4, we compare the techniques using explicit compaction and show the best-case speedup for each method. Data in Table 4 are based on a comparison to multi-threaded CSR when available, otherwise to the best proxy available in the respective paper.

7.2 Concurrent Symmetric SpMV

Several recent papers have also explored techniques for enabling concurrent symmetric SpMV. In addition to improving the CSB format by using register bitmasking, Buluç [7] describes a technique to exploit symmetric storage also using the CSB format. Starting from the observation that a block diagonal symmetric matrix is trivially parallelized, Buluç uses Reverse Cuthill-McKee (RCM) ordering to bring most non-zero elements close to the diagonal. The portions of the matrix, which are in blocks one and two positions off diagonal, are then assigned to separate rounds (coordinated to avoid write-write conflicts) while the elements further off diagonal (few in number) require atomic updates to the destination vector. The cluster scale solution, presented by Krotkiewski and Dabrowski [16], also starts by using RCM re-ordering and then handles all off-block-diagonal entries with local result buffers and attempts to overlap communication and computation across rounds. Such methods provide the benefit of scaling with threads, up to the memory bandwidth limit, but come at the cost of pre and post-processing steps and still require thread synchronization and either atomic updates or explicit communication.

Tangwongsan et al. [4], introduce a hierarchical storage format that points to the same memory region for off diagonal blocks if the matrix is symmetric (similar to our QTS format). These authors also point out that this kind of matrix storage enables lock and synchronization free concurrent symmetric SpMV. Their method achieves an average of 1.8x less memory traffic when using double precision values. These authors further reduce memory traffic by switching from double to single precision floating point, but do not use any other compaction or deduplication techniques.

8. CONCLUSIONS

In this work, we have shown that fine-grained data deduplication and DAG storage provide efficient representations for sparse matrices and can reduce memory traffic during the bandwidth limited SpMV kernel. Our results indicate that our method improves sparse matrix storage density by a factor of two on average (formats QTS, NZD, HCSR, and HCOO in Table 2). We show that the recursive nature of our data-deduplication technique can provide large advantages in storage density as demonstrated by the 5700x compaction achieved by the QTS format. Our worst-case storage density is $O(nnz)$ for a compressed sparse matrix (HCSR and HCOO) or $O(nnz \cdot log(n))$ for a logically dense or hybrid sparse matrix (RMA, QTS, and NZD).

Using our matrix storage techniques, we demonstrate an average speedup of 1.5x using a custom SpMV kernel running on standard hardware. By measuring DRAM accesses, we show a memory traffic advantage in proportion to the reduction in working set size. The relative benefit of our method depends on the memory traffic required (matrix specific) and the available bandwidth per core (Figure 12). For matrices exceeding LLC capacity (Figure 11), our data indicate an average reduction in memory traffic of 2.8x and best-case reduction of 90x.

For matrices with the highest potential speedup, instruction count overhead prohibits our software kernel from fully exploiting available memory bandwidth. The iterator register, described in the HICAMP architecture [9] (Section 2.3, Figure 10), eliminates this code overhead.

We introduce the novel QTS matrix format that inherently exploits any amount of matrix symmetry. Because the QTS format is *symmetry oblivious*, any thread safe work partitioning for the QTS format is safe for a symmetric matrix. Using this format, we demonstrate symmetric concurrent SpMV without any locks or matrix reordering.

Based on these results, we plan to implement iterator register hardware either as a custom feature in an existing ISA (e.g. using Tensilica TIE instructions [11]) or in an FPGA. For the specific purpose of SpMV, the iterator register functionality can be enhanced by adding a state machine to keep track of implicit matrix indices (for the QTS format) and a separate accumulate and store unit can take the fetch of destination vector elements off of the critical path.

As mentioned, current trends show that processing capa-

bility is increasing faster than memory bandwidth [1] and that caches are growing relatively faster than core count [5]. Both larger caches and additional cores benefit the method we have demonstrated. These trends and our data indicate that HICAMP enables otherwise difficult to achieve performance benefits.

9. REFERENCES

[1] A. Bechtolsheim. Technologies for Data-Intensive Computing. In *Proceedings of the 13th International Workshop on High Performance Transaction Systems*. HPTS, October 2009.

[2] M. Belgin, G. Back, and C. J. Ribbens. Pattern-based Sparse Matrix Representation for Memory-Efficient SMVM Kernels. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*, pages 100–109. ACM, June 2009.

[3] N. Bell and M. Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *Proceedings of Supercomputing (SC '09)*, pages 18:1–18:11. ACM, November 2009.

[4] G. Blelloch, I. Koutis, G. Miller, and K. Tangwongsan. Hierarchical Diagonal Blocking and Precision Reduction Applied to Combinatorial Multigrid. In *Proceedings of Supercomputing (SC '10)*, pages 1 –12, November 2010.

[5] S. Borkar and A. A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54:67–77, May 2011.

[6] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 233–244. ACM, 2009.

[7] A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication. In *International Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 721 –733, May 2011.

[8] D. R. Cheriton. Hierarchical Immutable Content-Addressable Memory Processor, January 2010. U.S. Patent 7650460.

[9] D. R. Cheriton, A. Firoozshahian, A. Solomatnikov, J. P. Stevenson, and O. Azizi. HICAMP: Architectural Support for Efficient Concurrency-Safe Shared Structured Data Access. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 287–300, New York, NY, USA, 2012. ACM.

[10] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38:1:1–1:25, November 2011.

[11] R. Gonzalez. Xtensa: A Configurable and Extensible Processor. *Micro, IEEE*, 20(2):60–70, Mar/Apr 2000.

[12] E.-J. Im, K. A. Yelick, and R. Vuduc. SPARSITY: Framework for Optimizing Sparse Matrix-Vector Multiply. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.

[13] K. Kourtis, G. Goumas, and N. Koziris. Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication Using Index and Value Compression. In *37th International Conference on Parallel Processing (ICPP '08)*, pages 511–519, September 2008.

[14] K. Kourtis, G. Goumas, and N. Koziris. Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression. In *Proceedings of the 5th Conference on Computing Frontiers (CF '08)*, pages 87–96. ACM, May 2008.

[15] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris. CSX: An Extended Compression Format for SpMV on Shared Memory Systems. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 247–256, February 2011.

[16] M. Krotkiewski and M. Dabrowski. Parallel Symmetric Sparse Matrix-Vector Product on Scalar Multi-Core CPUs. *Parallel Computing*, 36(4):181 – 198, 2010.

[17] M. Martone, S. Filippone, P. Gepner, M. Paprzycki, and S. Tucci. Use of Hybrid Recursive CSR/COO Data Structures in Sparse Matrix-Vector Multiplication. In *IMCSIT*, pages 327–335, October 2010.

[18] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[19] K. K. Nagar and J. D. Bakos. A Sparse Matrix Personality for the Convey HC-1. In *Proceedings of the 19th Annual Symposium on Field-Programmable Custom Computing Machines (FCCM '11)*, pages 1–8. IEEE, May 2011.

[20] S. Toledo. Improving the Memory-System Performance of Sparse-Matrix Vector Multiplication. *IBM Journal of Research and Development*, 41(6):711–725, November 1997.

[21] J. Treibig, G. Hager, and G. Wellein. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *Proceedings the International Workshop on Parallel Software Tools and Tool Infrastructures*, 2010.

[22] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[23] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *Proceedings of Supercomputing (SC '02)*, Baltimore, MD, USA, November 2002.

[24] J. Willcock and A. Lumsdaine. Accelerating Sparse Matrix Computations via Data Compression. In *Proceedings of the 20th International Conference on Supercomputing (ICS '06)*, pages 307–316. ACM, June 2006.

[25] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *Proceedings of Supercomputing (SC '07)*, pages 38:1–38:12. ACM, November 2007.