In-memory Data Compression for Sparse Matrices

Orion Sky Lawlor
Department of Computer Science
U. Alaska Fairbanks
lawlor@alaska.edu

Abstract

We present a high performance in-memory lossless data compression scheme designed to save both memory storage and bandwidth for general sparse matrices. Because the storage hierarchy is increasingly becoming the limiting factor in overall delivered machine performance, this type of data structure compression will become increasingly important. Compared to conventional compressed sparse row (CSR) using 32-bit column indices, compressed column indices (CCI) can be over 90% smaller, yet still be decompressed at tens of gigabytes per second. We present time and space savings for 20 standard sparse matrices, on multicore CPUs and modern GPUs.

1. INTRODUCTION

The future of high performance computing can be summarized as "communication dominates energy" [7]. The time and energy cost of sending a 32-bit value across the memory bus already exceeds the cost of floating point arithmetic, so increasingly, software can improve performance by spending arithmetic to minimize the data sent across the network, loaded from RAM, or stored in cache. For regular dense arrays, it is often sufficient to reduce the data size, for example from double to single to half precision floating point.

For irregular computations, such as sparse matrices, graphs, or unstructured finite element meshes [8], we face the difficult problem of referencing arbitrary neighbors. 64-bit pointers are large, and not easy to send via network, disk, or to the GPU. 32-bit array indices improve on each of these problems, and for sparse matrices, a list of 32-bit column indices per row is the common Compressed Sparse Row (CSR) format. In this paper, we propose further shrinking these to a variable bit per column format called Compressed Column Indices (CCI).

1.1 Sparse Matrix Dense Vector Multiply

A typical desktop-scale scientific problem today might have on the order of $n = 10^6$ unknowns, and involve the solution of a linear algebra problem like AX = B, where X and B are vectors and A is a matrix. The vectors are of dimension n and hence occupy a few megabytes each, which works fine in practice. The matrix A has a mathematical size of $n \times n$ entries, and a simple parallel CPU dense matrix-dense vector product might look like this.

```
// Dense matrix dense vector product.
void dense::product(real *src,real *dst)
{
#pragma omp parallel for
  for (int r=0;r<maxrow;++r) {
    real sum=0;
    for (int c=0;c<maxcol;++c)
      sum+=matrix[r][c]*src[c];
    dst[r]=sum;
  }
}
```

In practice the above code performs well for a small matrix, but dense storage quickly becomes problematic since $n^2 = 10^{12}$, so a dense matrix of this size would occupy several terabytes of memory, beyond the capacity of most desktops. Further, for many problems the vast majority of these matrix entries are zero, resulting in wasted space and time. For the matrices we examined, on average there are only 20 nonzeros per row, so in theory the matrix could be stored in about a hundred megabytes of RAM.

One solution to this problem is to abandon the mathematical abstraction of a matrix entirely, and compute the same vector product directly on the mesh or graph structure underlying the problem, which results in a "matrix-free method" [2]. These are frequently used for explicit finite element simulations, and can be made to work even for implicit iterative solvers including preconditioned Krylov-subspace methods [13]. However, because computing matrix-vector products this way involves problem-specific code written by a computational scientist working in a particular domain, it is labor-intensive to achieve high performance, especially on modern parallel computers.

A more performance-portable solution is to implement a sparse matrix, both as a common representation spanning problems and implementations, and as a clear dividing line between computational science and computer science. A sparse matrix explicitly lists the nonzero entries of the matrix, and never stores or computes the many zero entries, which allows a single implementation to achieve high performance on a wide variety of problems. There are several possible sparse matrix storage formats [1], but the most common is Compressed Sparse Row (CSR), so called because the zero values in each row are skipped over and effectively compressed out of each row. Removing zero values is a fantastic data compression method, and for a typical sparse matrix with 20 nonzeros per million entries, it saves about 99.996% of the space consumed by dense storage.

Proceedings of IA^3 2013 SC13, November 17–21, Denver, CO, USA

http://dx.doi.org/10.1145/2535753.2535758

1.2 Compressed Sparse Row (CSR)

In the existing sparse matrix format CSR, the matrix's nonzero values are packed into a single long 1D list named "val", the corresponding column number for each nonzero is stored in a second 1D list named "col", and each row's starting index into the previous two arrays is in a third list named "start". This code computes a sparse matrix dense vector product.

```
void csr::product(const real *src,real *dst)
{
#pragma omp parallel for schedule(dynamic,50)
  for (int r=0;r<maxrow;++r) {
    real sum=0;
    for (int i=start[r];i<start[r+1];++i)
      sum+=val[i]*src[col[i]];
    dst[r]=sum;
  }
}
```

For example, we will encode this 3×3 matrix in CSR form.

$$\begin{bmatrix} 9 & 5 & 0 \\ 0 & 8 & 0 \\ 6 & 0 & 7 \end{bmatrix}$$

In zero-based CSR format the three arrays will contain these values.

val=[9 5 8 6 7] matrix nonzero values (real)

col=[0 1 1 0 2] corresponding column index (int)

start=[0 2 3 5] each row's first index in arrays above

The last array has $n+1$ entries for an n-row matrix, and so is fairly small. The val and col arrays each have one element per nonzero in the matrix, and form most of the storage used by a typical sparse matrix.

Many have worked on compressing this column index array data. Willcock and Lumsdaine [14] provide a good summary as of 2006, though they did not address multicore or GPU, and could not get high performance without using assembly or other nonportable code. One aim of our work is to update this result for the multicore and GPU era.

Our particular data compression scheme will be described in the next chapter, but in general we use a variable bit-length differential code to compress the column numbers for each row into a packed "codes" array, and allow parallel access to any row by storing its first codes index in the "cstart" array. The remainder of the implementation follows the CSR pattern closely.

```
// CCI-format sparse matrix dense vector.
void cci::product(const real *src,real *dst)
{
#pragma omp parallel for schedule(dynamic,50)
  for (int r=0;r<maxrow;++r) {
    decoder_t cols(table,&code[cstart[r]]);
    real sum=0;
    for (int i=start[r];i<start[r+1];++i)
      sum+=val[i]*src[cols.next()];
    dst[r]=sum;
  }
}
```

The full code for the GPU decompressor is shown in the Appendix.

Library	Space Saved	1-CPU Rate for Decoding	Coding
CCI	92%	2.72 GB/s	0.42 GB/s
LZ4C	92%	0.72 GB/s	0.84 GB/s
zlib	96%	0.65 GB/s	0.08 GB/s
Gzip	95%	0.25 GB/s	0.16 GB/s
bzip2	96%	0.09 GB/s	0.01 GB/s

Table 1: Single-core data compression performance of our method of compressed column indices (CCI), and several existing libraries. Dataset is 32-bit matrix column deltas.

2. IN-MEMORY DATA COMPRESSION

Uncompressed CPU RAM can be accessed sequentially at tens of gigabytes per second, so any data compression algorithm used to replace a direct memory access must have a high rate. Because first decompressing all the data and then computing with it would send the uncompressed data on a round-trip to RAM, we must decompress and immediately use the data in place. Because CPU function call overhead (currently a few nanoseconds) would dominate this rate, we cannot even call a function for each decompressed data item if the items are on the scale of machine integers. One workaround could be to decompress one small cache-sized block of data at a time, which would work well with programmable scratchpad memory. On current hardware, it works best to combine the compression and data consumption functions, for example using runtime code generation, function inlining, or C++ templates.

As shown in Table 1, existing software data compression implementations are mostly well under a gigabyte per second per core. High performance software data compression libraries that forego bit-level operations can do better than this, such as the very recent LZ4 [3], which can exceed a gigabyte per second decompression rate with some datasets, although the rates suffer when data sizes are much less than a megabyte. A dedicated silicon implementation of the GZIP algorithm [12] showed an eightfold electrical energy reduction compared to CPU software, but still runs at only 0.31GB/s per chip.

For compressing matrix column indices, storing only the difference between adjacent columns or "delta coding" dramatically reduces the data size. In our experiments, the mean and median number of bits required for a raw column index were both 15 bits; delta coding reduced the mean to 5 bits and the median to 1 bit–over 50% of the columns are adjacent to the previous column. The table above shows compressed column index differences; compression savings are a few percent lower using non-delta coded columns, but speeds are similar.

2.1 Table-Driven Decompression

We compress matrix column indices using a typical variable bit length tree type coding scheme [6]. A naive implementation of this is particularly ill-suited to modern deeply pipelined CPUs due to the many data-dependent branch operations, which are inherently unpredictable, resulting in a high branch misprediction rate on CPUs. Modern GPUs do not perform branch prediction, yet GPU performance is still poor for typical compression trees, due to branch divergence and load/store divergence as threads in a single warp take data-dependent branches.

It is possible to structure a variable sized Huffman-type decom-

pressor so no data-dependent branch instructions are used during decompression, thus eliminating branch mispredict on CPUs and branch divergence on GPUs. This can be achieved using a data-dependent decode table [5], indexed by the next portion of compressed data, containing the decompressed value and distance to move ahead in the compressed dataset. Of course, for codewords of up to N bits, this requires a table with 2^N entries, which for reasonable column index maximum jumps between 16 and 32 bits, rapidly exceeds any plausible cache size. The table must be small enough to fit in cache because table accesses have low locality, so uncached table accesses would rapidly outweigh any time savings from data compression.

To reduce the size of the decode table, we borrow a technique from CPU instruction set encoding design, and separate codewords into opcode and immediate data portions. This allows our decode table to be sized to fit only the possible opcodes, typically 2–4 bits, and we can separately extract the immediate data from the code stream.

Variable-length opcodes can be supported using the simple technique of filling the table at a short opcode plus every possible data bit occupying the rest of the decode table index, a technique that appears to be well known among practitioners [10] but rarely discussed in literature. For example, suppose we use opcodes of up to two bits, so the decode table has four entries, but we only want the three opcodes `0`, `10`, and `11`. The two-bit opcodes `10` and `11` are simply written into the table at their index. Because the `0` opcode is only one bit long and so will be followed by a data bit when indexing the table, we fill in the decode table with two versions `01` and `00`, one for each possible value of the data bit.[1]

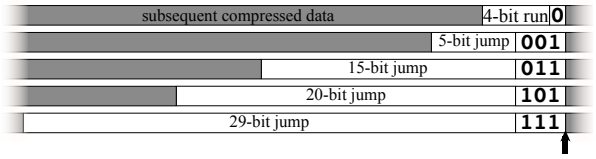

Figure 1: A compressed data item consists of an opcode, and the length of a contiguous run or distance to jump. Data is ordered from low to high bits, here right to left.

Figure 1 shows our column compression encoding, which uses at most 3 opcode bits. On the CPU, a leading opcode of `0` indicates a contiguous run of columns, with the run length encoded in the next four bits–experimentally, half all columns are adjacent to the previous column. On the GPU, due to thread stride within a row, contiguous runs are spread across threads, so the same leading opcode indicates a short jump of the magnitude of the thread stride. An opcode of `001` indicates a jump of up to 5 data bits, `011` indicates a 15-bit jump, `101` indicates a 20-bit jump, and `111` indicates a 29-bit jump, allowing a matrix of up to half a billion columns. Automated tuning for time and space indicates there is some variation in the optimum encoded bit allocation depending on the matrix, but using the same compression scheme for every matrix simplifies the software, costs negligible time, and costs only a few percent of space. All three-bit opcodes index into a decode table of 8 entries, where each entry is 8 bytes, resulting in a constant decode table that occupies 64 bytes, small enough to fit in a single read-only cache line.

[1]This is similar to expanding "don't care" entries into a full boolean truth table.

Another difficulty is most memory systems allow no access shorter than a byte, while our codes are arbitrary numbers of bits. One solution is to emulate bit pointers by loading a whole byte and shifting bits,[2] although this may only load a single valid bit for some indices. On x86 CPUs, we can efficiently load 64-bit values from unaligned addresses, so in a single load[3] we can extract at least 57 bits from an arbitrary bit location. GPUs do not support unaligned data access,[4] but they have enough threads to support a load loop efficiently, so this is less of a performance problem there. We look forward to when CPU instruction sets support a hardware SIMD gather operation, as has been promised for AVX2, since this would allow us to vectorize the simultaneous decompression of multiple rows.

3. THEORETICAL ANALYSIS

Consider an $n \times n$ sparse matrix, in which each row has an average of z nonzero entries. In an information theoretic [11] sense, CSR format uses at least $\mathrm{lb}\, n$ bits[5] to store the index of each column independently.

Because a column index can take any of n values, a single uniformly distributed column index indeed has Shannon entropy of $\lg n$ bits, so the above figure might appear optimal. However, in a sparse matrix each row has a list of unique columns, which can always be reordered into increasing order. The average distance between each neighboring pair of z such columns when distributed uniformly across n possible values is $n/(1+z)$, so the maximum average Shannon entropy of each column index jump is $\mathrm{lb}(n/(1+z)) = \mathrm{lb}(n) - \mathrm{lb}(1+z)$ bits. This is clearly smaller than the CSR format's $\mathrm{lb}\, n$ bits per index.

In any compression scheme, the uncompressed size minus the compressed size gives us the amount of space saved, and is often expressed as a percentage of the uncompressed size. Thus a scheme that saves 90% of space uses one tenth the original storage. Using the above analysis, the worst case space savings rate is a fraction of $\mathrm{lb}(1+z)/\mathrm{lb}(n)$. If the number of nonzeros per row z is a constant, the storage savings rate slowly approaches zero as n grows, which asymptotically puts the CSR format within an arbitrarily small factor of optimal. But if z grows with n by some constant factor f, such as $z = n/f$, the storage savings is approximately $1 - \mathrm{lb}(f)/\mathrm{lb}(n)$, which approaches unity as n grows, so the CSR format wastes an increasing fraction of storage.

4. EXPERIMENTAL ANALYSIS

We tested our data compression scheme on 20 matrices from the University of Florida Sparse Matrix Collection [4], which range from hundreds to millions of rows. They are listed in Table 2 sorted by the CSR storage space used, and show the percentage of this storage space saved by our column compressed index (CCI) storage scheme, and performance data for sparse matrix-dense vector products. Generally, the GPU methods are faster for matrices over a megabyte in size; for smaller matrices, the GPU suffers from kernel startup overhead, while a multicore CPU benefits from its cache.

The CPU used is an eight-thread four-core Intel Core i7-3770K at 3.50GHz, with 32GB RAM, using g++ 4.6.3 with -fopenmp, -O3,

[2]op=ptr[index/8]>>(index%8);

[3]op=(*(uint64_t *)(&ptr[index/8]))>>(index%8);

[4]Current CUDA hardware silently rounds unaligned accesses down to the nearest aligned address.

[5]We use the ISO notation for binary logarithm, $\mathrm{lb}\, N \equiv \log_2 N$

	Matrix		Space			CPU			GPU		
	Rows	Nonzeros	CSR	CCI	Total	GF/s		Time	GF/s		Time
Name	$\times 10^6$	z per row	MB	Saved	Saved	CSR	CCI	Saved	CSR	CCI	Saved
af_shell10	1.508	18.0	206.7	91%	30%	7.8	**9.6**	23%	15.1	**18.8**	25%
ldoor	0.952	24.9	181.1	92%	31%	7.0	**9.1**	30%	18.5	**20.5**	11%
msdoor	0.416	24.8	78.8	91%	30%	6.8	**8.7**	29%	16.5	**19.5**	18%
pwtk	0.218	27.2	45.2	94%	31%	8.1	**10.6**	31%	17.9	**22.5**	26%
hamrle3	1.447	3.8	42.1	65%	22%	**4.2**	3.8	-11%	**9.7**	6.0	-38%
hood	0.221	24.9	41.9	92%	31%	7.5	**8.7**	16%	14.9	**19.0**	28%
shipsec1	0.141	28.2	30.3	92%	31%	8.1	**8.9**	10%	17.1	**21.4**	25%
webbase-1M	1.000	3.1	23.7	55%	18%	**3.0**	2.8	-5%	**6.8**	4.6	-33%
consph	0.083	36.6	23.2	92%	31%	8.3	**9.6**	15%	18.3	**20.4**	11%
poisson3Db	0.086	27.7	18.1	72%	24%	**4.9**	4.2	-14%	**15.9**	7.9	-51%
pkustk10	0.081	27.2	16.7	92%	31%	8.1	**8.9**	9%	15.2	**19.4**	28%
pdb1HYS	0.036	60.2	16.7	96%	32%	8.5	**9.1**	7%	**25.1**	23.9	-5%
mc2depi	0.526	4.0	16.0	57%	19%	**5.6**	3.6	-36%	**9.0**	6.1	-33%
cant	0.062	32.6	15.5	93%	31%	8.5	**8.8**	3%	17.6	**22.6**	28%
scircuit	0.171	5.6	7.3	69%	23%	**5.0**	3.4	-31%	**7.7**	6.9	-10%
poisson3Da	0.014	26.1	2.7	74%	25%	**6.2**	4.5	-28%	**13.2**	9.3	-29%
thermal1	0.083	4.0	2.5	63%	21%	**4.3**	2.9	-33%	**6.3**	5.2	-17%
lock1074	0.001	24.5	0.2	94%	31%	**9.3**	6.9	-25%	4.0	**5.8**	45%
bcsstm38	0.008	1.0	0.1	5%	2%	1.5	**1.6**	6%	0.7	**0.8**	16%
poisson2D	0.000	6.6	0.0	69%	23%	**2.3**	**2.3**	0%	0.4	**0.8**	73%

Table 2: For each matrix, the matrix's total uncompressed size, the compression rate for column data alone, overall compression rate, and SMDV performance in billions of floating point operations per second (GF/s). We compare Compressed Sparse Row (CSR) with our Compressed Column Index (CCI) for both CPU and GPU. The highest performing method for each platform is in bold.

-msse3, and -ffast-math on Ubuntu Linux 12.04. Using the Intel compiler gave a slightly slower result than g++. Our CPU version is compared with CSR from the Intel Math Kernel Library 11.0 update 5, which is tuned for SIMD and multicore. The GPU is an NVIDIA GeForce GTX 570, using CUDA; our GPU CSR runtime comparison is with CUSPARSE 4.2.

4.1 Storage Space Analysis

Some applications are dominated by data transfer time, so nearly any transformation that saves space without dramatically inflating the computation time is beneficial [9]. Other applications are limited by RAM capacity, especially with the limited RAM available on the GPU, so again a space-time tradeoff would allow larger problems to be explored regardless of runtime cost.

As shown in Table 2, the achieved column index data savings rates with our CCI algorithm vary from above 90% for large highly regular matrices, to about 60% for large stochastic matrices, to under 10% for very small matrices with few nonzeros per column. These rates are a few percent less than those achievable with the dedicated compression libraries shown in Table 1, although those libraries' decompression rates are several times lower.

Based on the theoretical analysis in the previous section, if the column indices were uniformly randomly distributed—the worst case for data compression—we would expect a savings rate of $\mathrm{lb}(1 + z)/\mathrm{lb}(n)$. The average number of nonzeros per row is $z = 20$, so for an $n = 1000$ row matrix, this is 44%; while for $n = 1000000$ rows, this is 22%. In practice, our savings rates are much higher than this worst case, typically above 90% for $z > 20$ nonzeros per row, which indicates column indices have locality that allows them to compress well.

4.2 Performance Analysis: CPU

On a modern CPU, a naive table-driven implementation of CCI costs about 1 nanosecond per matrix nonzero, resulting in a delivered single precision sparse-matrix-vector performance of about 2 GFLOP per core. This is a poor result compared to a tuned CSR implementation such as the Intel Math Kernel Library's `mkl_scsrmv`, which delivers about 4 GFLOP per core. Yet CSR only scales to under 8 GFLOP on eight threads (2-fold scaling) due to the high memory bandwidth required.

The main limiting factor for using CCI on CPUs appears to be the single data dependency chain carried by the fetch-decode-shift portion of the decompressor's state variables. We can hide some of the latency along this chain by software interleaving several independent decompression streams. For example, we experimented with simultaneously decompressing several rows in a single loop, or dividing the columns of a single row into several independent streams separated by stride, in either case using several accumulators to break the dependency chain on `sum`. As shown in Figure 2, either technique improves performance somewhat, although naive CCI is actually higher performance for the highest thread counts. For comparison, we show the pointer-chasing List-of-Lists (LIL) data format, as an example of a format dominated by memory fetch latency.

Overall, CSR dominates for arrays small enough to fit in cache, but for large arrays where memory bandwidth becomes important, if CCI compression rates are good it outperforms CSR.

4.3 Performance Analysis: GPU

The GPU might seem like a less promising architecture for memory compression, due to its higher naive memory bandwidth, but the

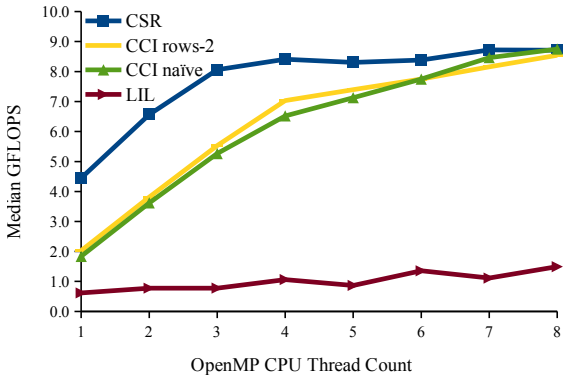

Figure 2: Multicore CPU scalability by data layout.

GPU's high thread count[6] allows it to tolerate latency much better than current CPUs. Currently, we do not attempt to parallelize the compression of sparse matrix column indices on the GPU, since the variable-length per row compacted output would require a two pass algorithm and parallel prefix traversal, and a GPU-side compression algorithm would require uncompressed data to be sent across the PCIe bus.

A direct, one thread per matrix row, CUDA version of our compressed column index decompression and sparse matrix dense vector multiply code runs at about 4 GFLOPS. A similarly naive version of CSR runs at just over 2 GFLOPS. Yet a well tuned commercial library implementation, like CUSPARSE's `Scsrmv`, can perform at over 20GFLOPS for the same matrices on the same hardware. The key for good performance on GPUs, as explained by Bell and Garland [1], is to use several threads per row, interleaving accesses to slice the row across GPU threads. This helps because interleaved matrix value and vector write accesses within a row are perfectly regular and aligned, while accesses between rows are necessarily ragged. Because the GPU executes adjacent threads simultaneously, better performance is achieved when adjacent threads are accessing nearby data at the same time; that is, GPUs do not reward per-thread locality, but across-thread locality.[7]

Using eight interleaved threads per row and compressed column indices, for matrices that compress well we can decompress sparse matrix column indices on the GPU at an aggregate rate of over 40GB/sec, and consistently deliver over 20GFLOP. For these matrices, compressed column index traversal performs up to 30% better than CUSPARSE 4.2.

Comparing CPU and GPU performance, we find the CPU dominates for small matrices due to the GPU's several microsecond kernel startup overhead. We have included the time to copy the result vector from the GPU to CPU, but ignored the time to copy the matrix itself from CPU to GPU, under the assumption that most scientific matrices are reused many times—for example, in the steps of an iterative Krylov solver, or an explicit timestepping approach. Including GPU matrix memory copy time skews the results further in favor of CCI due to its lower data volume.

[6]Typically thousands of active threads, from a pool of millions.

[7]A simple example: in a *row*-major 2D image, performance is much better if each GPU thread walks down a column, not across a row!

As shown in Figure 4, on both CPU and GPU, the mechanism by which CCI speeds up a matrix-vector multiply is by reducing the total memory bandwidth used. Although column indices are highly compressed, the vector and matrix values are uncompressed, resulting in a 33% maximum possible speedup. For favorable matrices, we are very close to this theoretical upper bound; for other matrices, the overhead of decompression outweighs the speedup from slightly lower memory bandwidth use.

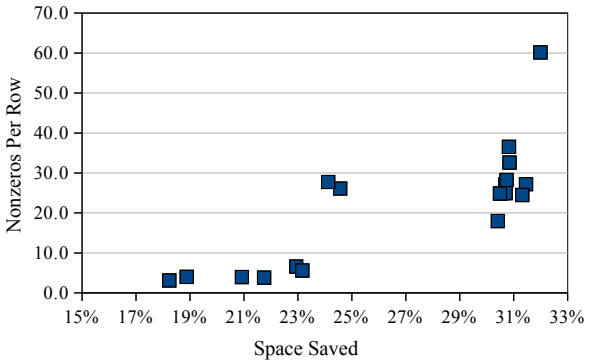

Figure 3: CCI's space savings improve dramatically if the number of nonzeros per row is high.

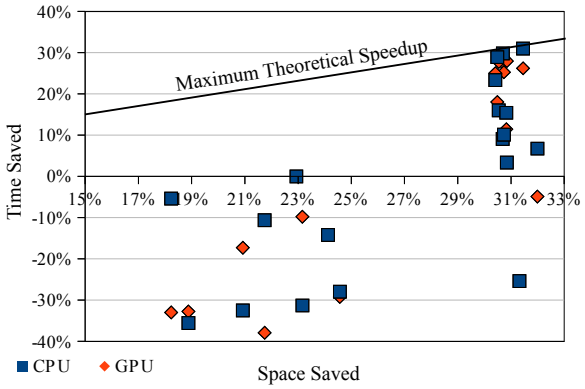

Figure 4: Comparing CCI's space savings versus its time savings. Negative savings indicate CCI is slower than CSR.

5. CONCLUSIONS AND FUTURE WORK

This work shows it is possible to save time by saving space. In-memory data compression on modern hardware is feasible, and can deliver just-in-time decompressed data at over 16 GB/s on multicore, and over 46 GB/s on the GPU. Although it is not always faster than CSR format, it is competitive with tuned commercial implementations.

Currently, we only compress column index data, but each matrix operation must also read the matrix and source vector values, and write the destination vector value, so if all four values have equal size the maximum theoretically achievable memory bandwidth improvement by compressing column indices is only 25%. Some matrix nonzero values are nearly incompressible, while others may be perfectly compressible (such as uniform), so compressing these would allow a speedup of up to 50%. A tile-based approach that caches source and destination vector values closer to the compute

elements could further reduce the overall memory bandwidth usage, although in all cases this depends on the compressiblity of the data.

Note that typically the $+$ and $*$ operations used in our matrix-vector product are floating-point hardware operations, but in general could form a semiring or an even simpler algebraic structure. Since we always do the multiplication before any additions, we do not even require distributivity over addition. This could be useful in graph algorithms, where "multiplying" a set of node values by the sparse adjacency matrix could perform a neighborwise operation such as minimum-cost-among-neighbors, which could be iterated to compute a minimum cost spanning tree in parallel.

Sparse matrices are applicable to a variety of problems, but they are also a common language in which a variety of applications can be expressed, and then transformed for high performance. We feel these runtime data storage transformations are a promising means to achieve high performance, but the real challenge is to find an interface that will allow applications to use them without extraordinary programmer effort.

6. REFERENCES

[1] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. *NVIDIA Technical Report NVR-2008-004*, 2008.

[2] P. N. Brown and A. C. Hindmarsh. Matrix-free methods for stiff systems of ODEs. *SIAM J. Numer. Anal.*, 23(3):610–638, 1986.

[3] Y. Collet. LZ4: Extremely fast compression algorithm. In *code.google.com*, 2013.

[4] T. A. Davis and Y. Hu. The university of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38:1–25, 2011.

[5] O. Edfors, P. O. Börjesson, A. Erendi, P. Ola, and B. S. A. Erendi. Analysis of a fast algorithm for look-up table based variable-length decoding. In *Proc. Radioveten. Konf*, pages 181–184, 1993.

[6] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the I.R.E.*, 40:1098–1101, 1951.

[7] S. W. Keckler, W. J. Dally, B. K. Khailany, M. Garland, and D. Glasco. GPUs and the future of parallel computing. *IEEE Micro*, 31(5):7–17, 2011.

[8] O. S. Lawlor, S. Chakravorty, T. L. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. V. Kale. ParFUM: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering With Computers*, 22(3):215–235, 2006.

[9] N. Reddy, R. Prakash, and R. M. Reddy. New sparse matrix storage format to improve the performance of total SPMV time. *Scalable Computing: Practice and Experience*, 13(2):159–171, 2012.

[10] M. Schindler. Practical Huffman coding. *http://www.compressconsult.com/huffman/*, August 1998.

[11] C. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, pages 379–423, July 1948.

[12] T. Summers. Hardware based GZIP compression, benefits and applications. In *AHA Products Group Whitepaper*, 2008.

[13] R. Telichevesky, K. S. Kundert, and J. K. White. Efficient steady-state analysis based on matrix-free krylov-subspace methods. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 480–484. ACM, 1995.

[14] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 307–316, New York, NY, USA, 2006. ACM.

Appendix: CCI Decode for GPU

```
// CCI-format product for GPU, in CUDA
enum {SLICES=8}; // threads per row
__global__ void CCI_product_device(
   const float *src,float *dst, // vectors
   int max_row, // size of matrix
   const int *start, // row index into vals
   const float *val, // matrix nonzeros
   const int *cstart, // code index per slice
   const code_t *code // compressed columns
   )
{
   uint idx=threadIdx.x;
   uint thread=idx+blockIdx.x*blockDim.x;
   uint row=thread/SLICES;
   uint slice=thread%SLICES;
   if (row>=max_row) return; // extra threads

   int col=-1; // last column number decoded
   // Location in compressed data array
   uint cindex=cstart[row*SLICES+slice];

   // Compressed data buffer (64 bits)
   unsigned long long m=0; // compressed data
   uint mbits=0; // count valid low bits of m

   // Loop over our slice of our row
   float sum=0.0;
   const int beg=start[row], end=start[row+1];
   for (int i=beg+slice;i<end;i+=SLICES)
   {
      // Refill compressed data buffer
      while (mbits<3*code_bits) {
         m=m|(code[cindex++]<<mbits);
         mbits+=code_bits;
      }

      // Low bits contain opcode
      const decode_table_t &t=
            decode_table[m&opcode_mask];

      // Higher bits contain data--
      // the difference in column numbers
      col+=t.mask & (m>>t.shift);

      // Remove consumed bits from buffer
      m=m>>t.count;
      mbits-=t.count;

      // Process this column number
      sum+=val[i]*src[col];
   }

   // Parallel-reduce row sum across slices.
   //  Each reduction is SLICES wide, but
   //  we may have more threads/block. See [1]
   __shared__ float sums[threads_per_block];
   sums[idx]=sum;
   if (slice< 4) sums[idx]+=sums[idx+4];
   if (slice< 2) sums[idx]+=sums[idx+2];
   // slice 0: write result to global memory
   if (slice==0) dst[r]=sums[idx]+sums[idx+1];
}
```