**AMATH 583 MIDTERM**

Ben Francis

**My own question:**

**Q**: Can both the row and columns be compressed, instead of either-or?

**A:** No. One immediate proof of this is that there is a continuous transition between the number of elements in "sparse" and "dense" matrices; there is no hard line between how many elements in a matrix make it sparse or dense.

As such, both dense and sparse representations need to be able to store O(n^2) elements.

The benefit of compression in \*either\* row or column storage is that we can make it go from proportional to the number of elements to proportional to the number of rows or columns, which is O(n), which is usually less than the number of elements. However, the other dimension is still proportional to the number of elements, so the amount of information needed to store the matrix values in the row/column indices is still O(n^2).

If both row and column were compressed, then the amount of information used to store the position of the matrix values would be O(n), since both row and column indices would be O(n).

If we steadily increase the number of elements in our sparse matrix until it is full, we can see that the value position storage is still O(n), which is not enough for the O(n^2) number of positions.

Then there is simply not enough information leftover if we were able to compress the rows and columns, no matter what code we used.

So the answer is no.

Note: Saying "never" to these type of things, especially in terms of code efficiency, I assume is either quite the slippery business, or would require a quite more sophisticated argument. With that said, I think the above constitutes a pretty good, first pass argument for why this cannot be done. This is often good enough to quench the curiosity, and to put the possibility of a doubly compressed sparse matrix storage in the "really really hard or maybe impossible" buck of problems. Knowing which problems are very hard or impossible is important in its own right.

**(1) How does the performance (in GFLOP/s) for sparse-matrix by vector**

**product compare to what you previously achieved for dense-matrix by**

**dense-matrix product? Explain, and quantify if you can, (e.g., using**

**the roofline model).**

First, we note that the results given are GLOP/s, NOT the total time. Sparse matrix-vector multiplication requires less flops overall, which gives them their benefit.

The best results here for sparse are the CSR/CSC family. These perform approximately at the same level of mult\_0 and mult\_1 for small inputs, which is the unoptimized multiplication and the hoisting optimization, and at a similar level as mult\_2, for larger inputs. which includes tiling. None of the methods can reach the performance of mult\_3, which has blocking.

A summation of the results is shown in the following table:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Mult\_0** | **Mult\_1** | **Mult\_2** | **Mult\_3** | **COO** | **CSR** | **CSC** | **AOS** |
| **128** | 1.69 | 1.99 | 6.55 | 18.27 | 1.09 | 2.03 | 1.86 | 1.17 |
| **256** | 1.13 | 1.33 | 4.57 | 17.57 | 0.95 | 1.7 | 1.47 | 0.9 |
| **512** | 0.88 | 1.26 | 4.27 | 15.83 | 1.00 | 1.44 | 1.39 | 0.79 |
| **1024** | 0.36 | 0.33 | 1.47 | 15.08 | 0.76 | 1.23 | 1.15 | 0.72 |

First, we notice that the sparse methods scale much better, with the exception of mult\_3. The blocking method in mult\_3 is presumably the reason for its ability to scale better. In the other dense methods, memory bandwidth will eventually prevent a high Flop/s count. Since the sparse methods have much smaller storage, this does not hurt them at scale as much.

After examining these empirical results, let us consider the numerical intensity of the methods. Prototypical analyses of dense matmat vs. sparse matvec have been undertaken in the Lecture 8 slides, namely in slides 75 and 77. It was found that the basic dense matmat algorithm has numerical intensity of 1/12 and the basic sparse matvec algorithm has numerical intensity of 1/14. These correspond to mat\_0 and COO.

Given a roofline model, these can then be converted into an expected GFlop/s count.

Let us then consider the roofline model that was found for my computer in homework 4.

We found the bandwidth of L1, L2 and DRAM, and the max flops of the computer:

Max: 53 GF/s

L1: 210 GB/s

L2: 115 GB/s

L3: 66 GB/s

DRAM: 38 GB/s

If one form of memory is being used, we can use its bandwidth information to calculate the GF/s we can get from our algorithm, depending on the numerical intensity. By the way, I am under the *impression* the above number are correct, pending successful implementation of the previous homework. I will proceed with them all the same, though in a sense the numbers are pretty exchangeable. If one number has been mistaken for another, or it is actually something different, or someone wishes to apply the following analysis to their own number, the values listed above should be thought of as adjustable parameters.

Each form of memory gives a slope, whose value is listed above for GB/s. To get GF/s, we take this value and multiply it by the numerical intensity, which is in units of F/B. Note that our units work out: F/B \* GB/s 🡪 GF/s.

Given 1/12 and 1/14 for dense matmat and sparse matvec numerical intensities, respectively, we can get the GF/s performance for each memory type:

|  |  |  |
| --- | --- | --- |
|  | **Dense** | **Sparse** |
| **L1** | 17.5 | 15 |
| **L2** | 9.6 | 8.2 |
| **L3** | 5.5 | 4.7 |
| **DRAM** | 3.2 | 2.7 |

A couple things should first be mentioned. It appears that the least straightforward part of constructing a roofline model is determining the y-intercepts associated with each memory type. In Lecture 8, slide 74 this is what leads to the result being 3 GFlops/s instead of 4.15, which is what is found by doing the (naïve) calculation 16.6/4. Note that all of the calculations above were done the naïve way, without incorporating the y-intercept. In lieu of an exact y-intercept, it can be noted that DRAM should have an approximate intercept in the range of ~ -1. All of the memory types should have an intercept between 0 and 1, except for L1 which might be a little over 1, but in the range of 1-2. Then while our above numbers are not exactly right, they are in the right range +/- 1. [Edit: I am now under the impression that there can’t be any “negatives”, and this discrepancy was due to something else. All the same, the y-intercept should only change the value on the order of 1GF/s. This isn’t trivial, but due to noise in our data and imperfections in our model anyway, doesn’t affect things too much. I will continue to reference the +/- 1, but keep this in mind as a potential edit).

Furthermore, the bandwidths of each cache are noisy. I ran the memory bandwidth experiment again, and got new answers:

L1: 186 GB/s

L2: 87 GB/s

L3: 65 GB/s

DRAM: 25 GB/s

With these new numbers, we can recalculate the above table:

|  |  |  |
| --- | --- | --- |
|  | **Dense** | **Sparse** |
| **L1** | 15.5 | 13.3 |
| **L2** | 7.2 | 6.2 |
| **L3** | 5.4 | 4.6 |
| **DRAM** | 2.1 | 1.7 |

Given the +/- 1, this can readily explain the sparse matrix multiplication COO results, from which the 1/14 numerical intensity was calculated, which was found to be between 0.75 and 1 GF/s. However, this only makes sense when considering DRAM. For mult\_0 and COO, the results seem to work if DRAM is being used, and incorporating the y-intercept in the range of -1, and the noisiness of the bandwidth measurements. Since we are using doubles, which each take 8 bytes, the 128 size matrices already take 8\*128\*128=131,072=128KB, which already exceeds L1. We repeat this calculation to get the storage involved with each dense matmat storage size:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Mat size | 128 | 256 | 512 | 1024 |
| Storage | 128KB | 512KB | ~2MB | ~8MB |

Sparse is less straightforward, since it is proportional to the number of non-zero elements.

From homework 4, we have the following cache sizes:

L1: 32 KB

L2: 256 KB

L3: 4MB

DRAM: 8GB

So matrix size 128 should be able to fit into L2, 256 and 512 should be able to fit into L3, and 1024 needs to use DRAM. When examining the empirical results from mult\_0, we do see these three “tiers”, though everything is scaled down. Again, we note that noisiness in most of the above assessments should be taken into consideration; this is to show some aspects of our model, not to make direct analytic calculations. For mult\_3, we see that it looks like L1 is being consistently used, with GFlops/s in the range of 15-18, compared to our predicted range of 15.5-17.5, which is good evidence for the efficacy of our calculations. The purpose of its blocking algorithmic optimization was to facilitate this cache benefit.

Now let us consider the relative results of dense matmat and sparse matvec. We wish to see if we can find a comparison between the 1/12 and 1/14 numerical intensities, accounting for the additional effects of caches. Since we only have the numerical intensities for mult\_0 and COO, let us compare them side-by-side, and their ratios:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Mult\_0** | **COO** | **Ratio** |
| **128** | 1.69 | 1.09 | 1.55 |
| **256** | 1.13 | 0.95 | 1.18 |
| **512** | 0.88 | 1.00 | 0.88 |
| **1024** | 0.36 | 0.76 | 0.47 |

In order to compare mult\_0 and COO, we need to finally consider the size of the storage COO is working with. This is not proportional to the matrix size, but rather to the number of non-zero elements (which is indirectly proportional to matrix size). Since COO will have three arrays of this size, we multiply by 8 bytes to account for the double and 3 to account for the three storage arrays to get the total size used by COO. Since we do not know the number of non-zero elements, let us check both a “linear” assumption of sparsity (number of elements equal to size of matrix, so 128 elements in a 128x128 matrix, and then assume there are no zeros at all, to get a worst case scenario estimate. Then we get [mat\_size] \* 3 \* 8 to get the best case storage size and [mat\_size]^2 \* 3 \* 8 to get the worst case storage size.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Best | Half | Worst |
| 128 | ~3KB | ~191KB | ~382KB |
| 256 | ~6KB | ~768KB | ~1.5MB |
| 512 | ~12KB | ~3M | ~6MB |
| 1024 | ~24KB | ~12MB | ~24MB |

In the worst case scenario, the storage is three times worse than dense, and the numerical intensity is 12:14 worse, so there is no win from doing sparse. The best-worst case spectrum should be linear, proportional to the percentage of how many elements there are per row relative to the number possible. So for 128, the best case scenario is 1/128, and worst is 128/128. Then for 50% capacity of elements, it would be ~382 \* 0.5 = ~191. These are also shown in the table. This does provide a pretty large sweep, from the best case all fitting into L1 to the worst case fitting into L3, L3, DRAM, and DRAM, respectively. For the half capacity, we get L2, L3, L3, and DRAM.

With this new analysis, let us come back to the observed ratios. From the 1/12 : 1/14 ratio alone, we expect mult\_0 to perform 1.16 times faster. This is observed basically exactly for the 256 case, and if you average the first three sizes’ ratios (1.55, 1.18, and 0.88), we do recover 1.2, which is close to 1.16. However, it is the nature of the sparse matrix to scale better, given less-than-dense element capacity in the matrix. We observe the ratio shifting to 0.47 with size 1024. Of course, the trend was downward from the beginning, but this seems to be a phase shift into a different regime where the COO is significantly better.

A note on the above: I am seeing a piazza post (“Lecture 8 question”, question @99) that mentions the unhoisted dense matmat may have 1/16 numerical intensity, due to 4 additional operations, which in turn would make the sparse 1/18. This would shift our ratio from 1.16 to 1.12. This is not all that large of a change. I will, however, still with the 1/12 and 1/14, since I can cite them directly from Lecture 8 slides 75 and 77. However, it is worth noting the above discussion may be slightly off for this reason. With all that said, the calculations and reasoning remain exactly the same, so it is only a slight shift in a parameter.

**(2) Referring to the roofline model and the sparse matrix-vector and dense matrix-matrix algorithms, what performance ratio would you theoretically expect to see if neither algorithm was able to obtain any reuse from cache?**

With no cache help, (and thus fixed bandwidth for both) the numerical intensity alone will dictate performance. Since sparse matvec has 1/14 numerical intensity and dense mat has 1/12, we expect the dense mat to have better performance by a ratio of 1/12 : 1/14, which is 1.16X better. This has been described above as well, and empirically examined incorporating the measured results from mult\_0 and COO.

If, as mentioned above, the actual numerical intensities should be 1/16 and 1/18, then the non-cache performance ratio should be 1/16 : 1/18 🡪 1.125

**(3) How does the performance (in GFLOP/s) for sparse-matrix by vector**

**product for COO compare to CSR? Explain, and quantify if you can,**

**(e.g., using the roofline model).**

From our earlier table, now only examining COO and CSR, we see:

|  |  |  |
| --- | --- | --- |
|  | **COO** | **CSR** |
| **128** | 1.09 | 2.03 |
| **256** | 0.95 | 1.7 |
| **512** | 1.00 | 1.44 |
| **1024** | 0.76 | 1.23 |
| **2048** | 0.98 | 1.43 |

What accounts for this difference? CSR takes less storage, where now the storage is approximately 2 \* [NNZ] \* 8 + [mat\_size] \* 8. This is the same as COO, but one of the arrays goes from being proportional to the number of non-zero elements to being proportional to the matrix size; this is the “compression. This gives it approximately (depending on the number of non-zero (NNZ) elements) 2/3 the storage.

Let us now comment on the numerical intensity of the CSR matvec algorithm, compared to COO, although this is tricky. First, I have used hoisting in my algorithm, and as such this will be incorporated in the assessment. CSR gets a for loop, which should/could hurt the numerical intensity (right? If not, ignore this. Not sure if compiler can help remove the added cost of maintaining the for loop variable, perhaps by inlining it). Now let us examine what really hurts COO’s numerical intensity. It has to read into row\_indices to get the index for y. Since it cannot hoist, this also hurts it since this has to be done every time. Note that the degree to which hoisting helps depends on the distribution of the non-zero elements. Even without hoisting, in CSR we only have to read into y with the *i* loop variable, instead of getting it from reading into row\_indices. Examining just the interior loop (this doesn’t account for reading and writing the hoisting variable when needed), our comparison between COO and CSR is:

(I’ll use the following shorthand: R 🡪 Read, W 🡪 Write, A 🡪 Add, M 🡪 Multiply, I🡪 Integer, D🡪 Double. So RD is “Read Double”).

**COO (Loop interior)**

y(row\_indices[k]) = y(row\_indices[k]) + storage[k] \* x(row\_indices[k])

RI, RD. WD RI, RD AD RI MD RI, RD

Read/Writes: 8

Flops: 2

**CSR (Loop interior)**

t = t + storage[k] \* x(row\_indices[k])

WD AD RI MD RI, RD

Read/Writes: 4

Flops: 2

This shows an improvement of 100%, though this is again only considering the inner part of the loop. The hoisting will hurt this, to a degree depending on the distribution of non-zero elements in the matrix.

I do have a slight uncertainty if we have to RI/RD twice for the +=; so that might be too much. This is notably different than the 1/14 we calculated for COO in the lecture slides, so this is for sure not exactly right. This is meant to provide a *heuristic*, to show how the hoisting and savings on reading y with an index instead of reading into row\_indices to get the index saves numerical intensity.

Also, we eventually have to pay the piper and to the read/writes for the hoisting; as mentioned, the degree to which this saves us is dependent on the sparsity of the matrix, since the hoisting only helps when we’re moving within a row. It could even hurt us, in the cases of extreme sparsity.

All of this is to say it is a difficult calculation, but in most sparse matrices the hoisting will help us in CSR, and the single read into y (instead of reading into an array to get the index for y) will also help us in CSR. CSR also has better storage.

Empirically, we see the CSR performs approximately 100% (twice) as well for some sizes, and around 150% better for other sizes. Pinpointing this down is difficult, especially since the ratio of performance is not consistent.

The ratios CSR:COO for each of the sizes are:

128 🡪 1.86

256 🡪 1.78

512 🡪 1.44

1024 🡪 1.61

2048 🡪 1.45

Our analyses of the numerical intensity and storage are pretty encouraging in interpreting this; we got 100% numerical intensity improvement in the inner for loop, but didn’t incorporate other things like the hoisting costs. The storage is helped, but this is proportional to the number of non-zero elements, so it is difficult to determine its exact consequences. There will be certain sizes and sparsities of matrices where this could cause a “jump”, wherein the matrix could fit in a lower sized caches. The empirical result is somewhere inbetween this super-idealized doubling of performance and a more realistic similarity of performance, taking into account that the storage savings may not let it fit into new caches, and that the hoisting effects may not help as much as anticipated. The “one read” into y, instead of “two reads” should provide the most consistent benefit, since even without hoisting this will save a read operation that will return an integer.

**(4) How does the performance (in GFLOP/s) for sparse-matrix by dense**

**matrix product (\*\*SPMM\*\*) compare to sparse-matrix by vector product**

**(\*\*SPMV\*\*)? The performance for SPMM should be about the same as for**

**SPMV in the case of a 1 column dense matrix. What is the trend with**

**increasing numbers of columns? Explain, and quantify if you can,**

**using the roofline model.**

Let us compare these for COO, CSR, and CSC, though we will do the bulk of our analysis making our assumptions (whatever they are) with COO, since it is simplest and most archetypal of sparse matrix operations. (Aside: Why haven’t I been mentioning AOS more? Answer: Because it is not very good, and not particularly insightful. CSR and CSC at least have the benefit of being state of the art, and showing how *better* performance than COO acts. )

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **MatVec (SPMV)** | | | **MatMat (SPMM)** | | |
|  | **COO** | **CSR** | **CSC** | **COO** | **CSR** | **CSC** |
| **64** | 1.30 | 2.32 | 2.70 | 0.94 | 2.32 | 2.06 |
| **128** | 1.32 | 2.25 | 2.67 | 1.16 | 2.33 | 2.01 |
| **256** | 1.18 | 2.09 | 2.18 | 1.10 | 2.03 | 1.74 |
| **512** | 1.15 | 1.79 | 1.43 | 1.12 | 1.83 | 1.55 |
| **1024** | 0.86 | 1.46 | 1.53 | 0.99 | 1.48 | 1.38 |
| **2048** | 0.98 | 1.43 | 1.23 | 0.94 | 1.43 | 1.29 |

Here we can see that the values are actually pretty much the same. This is expected, since matmat mult is like many matvec mults. Now that we have shown that matvec is like the degenerate case of matmat, we consider the question:

“What is the trend with increasing number of columns?”

This has been discussed in previous sections, though I will review it as well (it is my hope that the grader/reader will take the previous discussions into account for this).

The performance degrades over time (for both matmat and matvec); this is expected since it is progressively taking up more storage. As has been mentioned previously, the amount of storage depends approximately linearly on the sparsity of the matrix. The best, middle, and worst case scenarios have been given for this in Question 1. Now that we have shown that SPMM scales identically according to SPMV, we have the same result.

Initially, I began looking at the results to see which jumps in performance were “linear” versus “drop-offs”, to seek where a cache size was exceeded. However, one can notice that COO goes from 0.86 to 0.98 after doubling its size. This cannot be really theoretically accounted for, and so just provides us evidence that there can be sizable noise in the data. Still, one can see in multiple of the columns that sometimes there are jumps of size ~0.1 GF/s, and sometimes there are jumps of size ~0.3. The latter seem like better candidates to be changes in which cache are being used.

An interesting question is why the storing of the *other* matrix that is being multiplied, the “B” matrix in A x B = C, where A is the sparse matrix we are operating on, does not cause cache problems. I would thing that A x b = c, where b is a vector, would be more readily read, and that reading B would cause cache problems. However, empirically we do not notice any dropoff from this. The probable answer I can think of is that the “vector” column of B used to calculate each column of C (this is what makes matmat much like independent matvec) does not incur much overhead when it needs to be switched out for the next vector/column of B. Within each column of B, they appear exactly identical to matvec with Axb. The only need is to switch out b\_1 (the first column of b) with b\_2 (the second column of b) when we are done with b\_1; evidently this swapping of columns does not take too much read/write. Either way, it should not take up any *more* space, which answers my question as to why this does not cause additional cache problems. Notice the columns are never reused after we have computed their corresponding column in C. Question self-answered!

**(5) How does the performance of sparse matrix by dense matrix product (in**

**GFLOP/s) compare to the results you got dense matrix-matrix product in**

**previous assignments? Explain, and quantify if you can, using the**

**roofline model.**

The following table summarizes these results:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Mult\_0** | **Mult\_1** | **Mult\_2** | **Mult\_3** | **COO** | **CSR** | **CSC** | **AOS** |
| **128** | 1.69 | 1.99 | 6.55 | 18.27 | 1.16 | 2.33 | 2.01 | 1.29 |
| **256** | 1.13 | 1.33 | 4.57 | 17.57 | 1.10 | 2.03 | 1.74 | 1.04 |
| **512** | 0.88 | 1.26 | 4.27 | 15.83 | 1.12 | 1.83 | 1.55 | 0.94 |
| **1024** | 0.36 | 0.33 | 1.47 | 15.08 | 0.99 | 1.48 | 1.38 | 0.83 |

Notice that the left hand side is the same as was examined in Question 1.

This analysis is actually approximately the same as was conducted in Question 1, since in Question 4 we have examined and explained why SPMM is computationally and performance-wise almost identical to SPVM, with the only difference being the exchanging of columns of B when we have gone through the previous column, with no reuse. This exchanging will only get linearly worse with scaling of the matrix, since we only need to move one *column* at a time. So I refer the reader to the exact same discussion that was undertaken in Question 1 ☺ (which, in accordance with the principle DRY, I will not repeat here!). I suspect that realizing this approximate equivalence is the real point of this question anyway.

**Extra Credit**

I added hoisting to CSR and CSR^T, as well as CSC and CSC^T. For CSR and CSC^T, this is the normal hoisting t = y(i).

However, for CSR^T and CSC, we cannot do this, since y depends on j. Instead, we hoist t = x(i). Because of this difference, we never reassign t to anything; this hoist only prevents repeated reads to x(i).

I also hoisted for CSR and CSC matmat, where CSR got the hoist t = C(j,i), and after the inner loop assigned C(j,i) = t, and where CSC got the hoist t = B(j,i), with no reassigning.

(I can’t remember if CSR already came with hoisting … if it did, then I used it for CSC!)

**One thing I learned from this assignment was …**

I think I came into this with a much more topical understanding of sparse matrices, and in particular how the compressed sparse matrix multiplication and storage really worked. Having to really work through the algorithm myself really cemented my understanding with it. I had to really think through what was happening with the information in a mechanical way, which was very helpful.

TL;DR: Compressed sparse matrix multiplication

**One thing I am still not clear on …**

How to practically calculate the y-intercepts in the roofline model.