Student Name: Rohan Tiwari
AMATH 583 Midterm

Sparse-Matrix Vector Product Performance

---

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).**

Sparse-matrix by vector product was run multiple times on the computer. Given below is a sample run for unoptimized implementations:

N(Grid)	N(Matrix)	NNZ	COO	COO^T	**CSR**	CSR^T	CSC	CSC^T	AOS	AOS^T
64	4096	20224	0.935218	0.939609	**3.45063**	0.966844	0.957592	3.33561	0.971537	0.957592
128	16384	81408	0.920134	0.924375	3.34316	0.95066	0.964372	3.34316	0.937333	0.937333
256	65536	326656	0.705668	0.733792	1.26579	0.847392	0.836887	1.09474	0.672846	0.688866
512	262144	1308672	0.722026	0.781297	1.22449	0.844305	0.802251	1.24635	0.662619	0.673272
1024	1048576	5238784	0.710867	0.710867	0.662044	0.424268	0.613191	1.16417	0.575141	0.584913
2048	4194304	20963328	0.753738	0.750365	1.14086	0.722873	0.729159	1.1606	0.450824	0.462002

Table 1: Sparse-matrix vector multiplication (unoptimized)

CSC^T is the best performing sparse-matrix product for most problem sizes except for 256 and 64 in the above sample which are more efficient with CSR. Overall, CSR and CSC^T perform similarly.

From PS4, the following was a sample run of the dense-matrix product

N	mult_0	mult_1	mult_2	**mult_3**	mul_t_0	mul_t_1	mul_t_2	mul_t_3
8	3.35502	3.48922	8.72305	14.5384	4.84614	3.35502	8.72305	10.9038
16	3.17202	3.32307	8.02119	20.5248	3.61577	3.33897	8.30766	15.8601
32	2.59628	2.61124	7.74445	**29.2291**	2.82274	2.61124	8.09019	23.8448
64	2.18559	2.37439	6.5726	27.4854	2.21224	2.46472	7.14188	20.6141
128	1.63077	1.81318	6.61311	25.3632	1.88946	1.95633	6.77947	17.8171
256	1.02385	1.23513	5.84325	22.8309	1.75067	1.78596	6.61072	16.1649
512	0.240609	0.338325	1.62093	21.5795	1.61945	1.64822	6.27808	14.801

Table 2: Dense-matrix dense-matrix multiplication

Student Name: Rohan Tiwari
AMATH 583 Midterm

For dense-matrix product, generally the GFLOPs/s is higher than sparse-matrix by vector product for optimized versions. Mult_3 was the best performing with hoisting, tiling, blocking and loop reordering (ikj) optimizations.

Looking back at PS4, the maximum compute performance of the computer is 44.8 GFLOPS/s.

Maximum achieved performance for dense-matrix product is for mult_3 = 29.2291 GFLOPS/s with a problem size of 32. Considering 4.2 GHz (boost speed), we get 29.2291/4.2=6.96 Flops/cycle. This is for a mult_3 implementation with (i,k,j) loop ordering. The problem size of 32 can easily fit in the L1 cache (512KB) as problem size of 32 is 32*32*8= 8192 bytes.

Let us look at the arithmetic intensity of this mult_3 implementation. This implementation's inner loop has 8 flops and accesses 4 doubles in memory (8*4 = 32 bytes). This gives an arithmetic intensity of 8/32 = 0.125 flops/byte. We check the roofline (docker) model (L1 cache) and see that this corresponds to roughly 30 GFLOPS/s which matches closely with the performance of mult_3. See Figure 1 below for roofline from PS4 (docker). Please note that L2, L3 are disabled on docker on this machine.

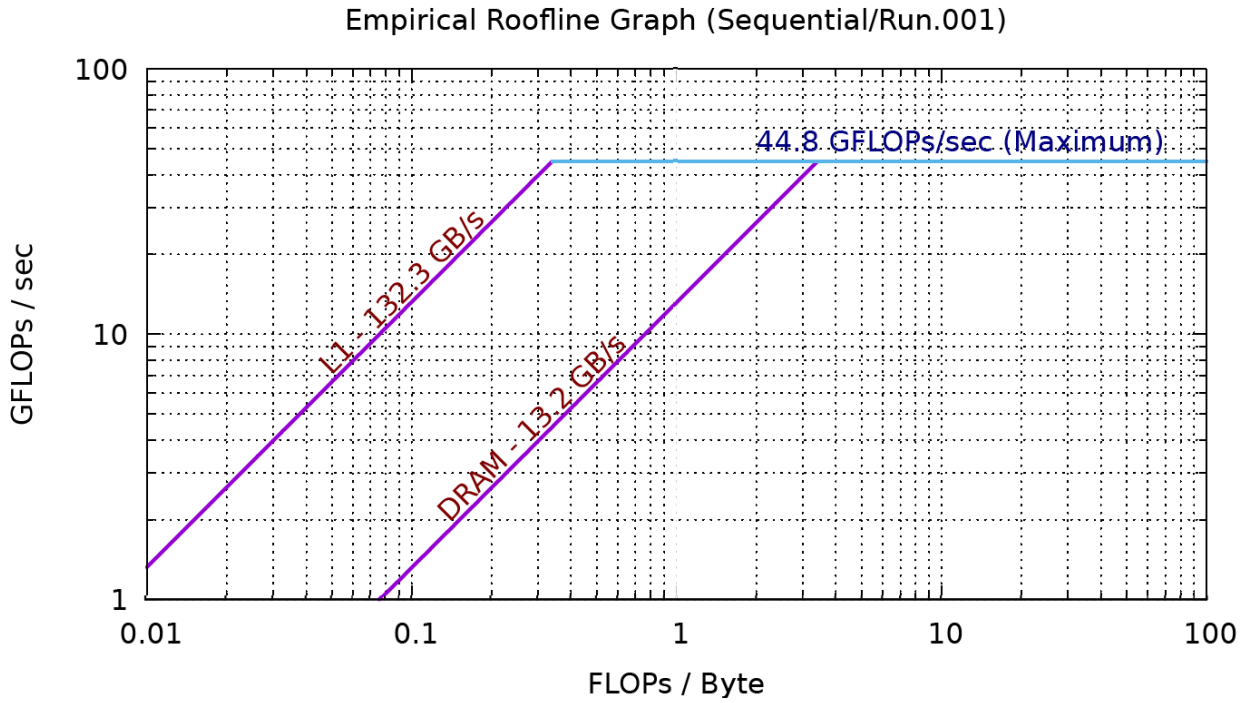

Figure 1: Roofline from PS4 for this machine (Docker). Please note that L2, L3 are disabled on docker on this machine as per piazza discussion.

It is worth noting that the storage space required by sparse-matrix is much lower than dense-matrix. The space savings are achieved by storing sparse matrices in optimized formats starting from COO and AOS to more optimized CSR and CSC formats.

Maximum achieved performance for sparse-matrix vector product is for CSR = 3.45063GLOPS/s with a problem size of 64. Considering 4.2 GHz (boost speed), we get 3.45063 / 4.2 = 0.82157 Flops/cycle which is considerably lower than the best performing mult_3 implementation above. This problem size can fit

in the L1 cache (512KB) like the best performing mult_3 implementation because NNZ=20224 and therefore total size = 20224*8 = 161792bytes = 161.792Kb. The space required to store indices in uncompressed form is 20224*8 = 161.792kb (8 bytes is for 2 int row/col indices). The total of roughly 322kb can easily fit in L1 cache as CSR also compresses the row indices. The inner loop in the unoptimized implementation has 2 flops and accesses 3 doubles and 1 int (8*3 + 4 = 28 bytes) which is an arithmetic intensity of 2/28 = 0.07 flops/byte which should correspond to ~10 GFlops/s from roofline model. The sparse-matrix vector multiplication has low arithmetic intensity as compared to the dense matrix multiplication.

From roofline model, ratio of the estimated performance of best performing implementation of sparse-matrix and dense-matrix multiplication is 10/30 = 0.33. Comparing this to the actual performance we get 3.45063/29.2291 = 0.11805 which is much lower than the expected ratio from roofline perhaps indicating that sparse-matrix vector implementation can be optimized further.

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?**

For dense-matrix multiplication, an arithmetic intensity of 0.125 flops/byte gives 2 GFlops/s considering the DRAM line on the roofline model as it is unable to obtain reuse from cache.

For sparse matrix vector multiplication, an arithmetic intensity of 0.07 flops/byte gives almost ~1 GFlops/s considering the DRAM line on the roofline model as it is unable to obtain reuse from cache.

Ratio of sparse matrix vector theoretical performance to dense-matrix theoretical performance is 1/2 = 0.50.

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).**

Sparse matrix by vector product was run multiple times on the computer. Given below is a sample run for unoptimized implementations:

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T	AOS	AOS^T
64	4096	20224	0.935218	0.939609	3.45063	0.966844	0.957592	3.33561	0.971537	0.957592
128	16384	81408	0.920134	0.924375	3.34316	0.95066	0.964372	3.34316	0.937333	0.937333
256	65536	326656	0.705668	0.733792	1.26579	0.847392	0.836887	1.09474	0.672846	0.688866
512	262144	1308672	0.722026	0.781297	1.22449	0.844305	0.802251	1.24635	0.662619	0.673272
1024	1048576	5238784	0.710867	0.710867	0.662044	0.424268	0.613191	1.16417	0.575141	0.584913
2048	4194304	20963328	0.753738	0.750365	1.14086	0.722873	0.729159	1.1606	0.450824	0.462002

Table 3: Sparse-matrix vector multiplication (unoptimized)

CSR performs much better than COO on smaller problem sizes but their performance approaches each other at large problem sizes e.g. at 2048 the difference is reduced but CSR is still more efficient. The ratio starts at almost 3.5 for COO versus CSR but drops to around 1.5 for larger problem sizes.

CSR implementation uses less data than COO implementation as the row_indices array is compressed and there are better chances of it fitting in closer (smaller) caches increasing locality.

Looking at CSR unoptimized implementation, we see it accesses 3 doubles and 1 int (total of 28 bytes) and performs 2 flops in the inner loop giving an arithmetic intensity of 0.07 flops/byte. COO implementation also accesses 2 ints and 3 doubles (8*3 + 2*4 = 32 bytes) and performs 2 flops giving an arithmetic intensity of 0.0625 flops/byte. Looking at best performing CSR and COO implementations at problem size 64, we see that they fit in the L1 cache. The corresponding bandwidth numbers for CSR are ~10 GFlops/s and for COO are ~8 GFlops/s. The ratio of theoretical performance is ~0.8.

Since the row indices array is compressed for CSR, it allows faster access to row indices which gives higher Gflops/s.

Sparse-Matrix Dense-Matrix Product

---

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.**

Given below is a sample run of matmat.exe to print results of SPMM using unoptimized versions

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC	AOS
64	4096	20224	1	0.667392	3.22302	0.674202	0.674202
128	16384	81408	1	0.649668	3.08214	0.665991	0.637174
256	65536	326656	1	0.575138	1.24613	0.628889	0.480651
512	262144	1308672	1	0.581632	1.11289	0.572213	0.362401
1024	1048576	5238784	1	0.343527	0.725719	0.38187	0.546062
2048	4194304	20963328	1	0.606377	1.23314	0.440671	0.405929

Table 4: Sparse-matrix dense-matrix multiplication (unoptimized)

SPMM has the best numbers for CSR. SPMV had best numbers for CSC^T but CSR was close. We can conclude that SPMM and SPMV performance is similar for CSR after checking the sample runs in tables 3 and 4. As the problem size increases, the performance generally drops but interestingly for both SMMV and SPMM CSR implementations there is a slight increase for size 2048 after 1024.

Looking at SPMV CSR unoptimized implementation, we see it accesses 3 doubles and 1 int (total of 28 bytes) and performs 2 flops in the inner loop giving an arithmetic intensity of 0.07 flops/byte. Looking at

SPMM CSR unoptimized implementation, we see it accesses 3 doubles and 1 int (total of 28 bytes) and performs 2 flops in the inner loop giving an arithmetic intensity of 0.07 flops/byte. Since the arithmetic intensities are similar, we expect same theoretical performance for CSR on roofline (Figure 1) of about ~10GFlops/s which would give a theoretical performance ratio of 1.0. The actual numbers are not far off e.g., the actual performance ratio for SPMV and SPMM for size 64 is 1.07.

Comparing COO and AOS for SPMM and SPMV, we notice that both implementations perform better for SPMV. SPMV to SPMM ratio tends to be around 1.4-1.1 for different problem sizes for AOS and COO. Arithmetic intensity for COO SPMM 2/32 = 0.0625 flops/byte (2 flops and 3 doubles + 2 ints in inner loop) which is same for COO SPMV. Size 64 for both COO SPMM, SPMV fits in the L1 cache and with same arithmetic intensity should give theoretical performance ratio of 1.0 using roofline. Same can be said about AOS implementation of SPMM and SPMV.

Comparing CSC for SPMM and SPMV, the performance for SPMV is better. The actual performance ratio is between 1.4-1.6 for different problem sizes. Arithmetic intensity for CSC SPMV is 2/28 = 0.07 flops/byte (3 doubles and 1 ints with 2 flops) same as CSC SPMM which should lead to the same theoretical performance ratio of 1.0 using roofline and problem size 64.

Generally, with larger problem sizes (increasing number of columns) the ratio does not tend to get much worse between SPMM and SPMV.

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.**

From PS4, the following was a sample run of the dense-matrix product

N	mult_0	mult_1	mult_2	mult_3	mul_t_0	mul_t_1	mul_t_2	mul_t_3
8	3.35502	3.48922	8.72305	14.5384	4.84614	3.35502	8.72305	10.9038
16	3.17202	3.32307	8.02119	20.5248	3.61577	3.33897	8.30766	15.8601
32	2.59628	2.61124	7.74445	**29.2291**	2.82274	2.61124	8.09019	23.8448
64	2.18559	2.37439	6.5726	27.4854	2.21224	2.46472	7.14188	20.6141
128	1.63077	1.81318	6.61311	25.3632	1.88946	1.95633	6.77947	17.8171
256	1.02385	1.23513	5.84325	22.8309	1.75067	1.78596	6.61072	16.1649
512	0.240609	0.338325	1.62093	21.5795	1.61945	1.64822	6.27808	14.801

Table 5: Dense-matrix dense-matrix multiplication

Given below is a sample run of matmat.exe to print results of SPMM using unoptimized versions

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC	AOS
64	4096	20224	1	0.667392	**3.22302**	0.674202	0.674202
128	16384	81408	1	0.649668	3.08214	0.665991	0.637174
256	65536	326656	1	0.575138	1.24613	0.628889	0.480651

512	262144	1308672	1	0.581632	1.11289	0.572213	0.362401
1024	1048576	5238784	1	0.343527	0.725719	0.38187	0.546062
2048	4194304	20963328	1	0.606377	1.23314	0.440671	0.405929

Table 6: Sparse-matrix dense-matrix multiplication (unoptimized)

The dense-matrix product from PS4 is considerably more efficient. Mult_3 uses hoisting, tiling, blocking and loop ordering optimizations. The ratio between best performance of Mult_3 and best performing sparse-matrix product (CSR) is almost 10-20 for larger problem sizes (64 and above).

Looking back at PS4, the maximum compute performance of the computer is 44.8 GFLOPS/s. We compare the best performing CSR (problem size 64) SPMM implementation with the mult_3 dense implementation for the same problem size.

Achieved performance for dense-matrix product is for mult_3 = 27.4854 GFLOPS/s with a problem size of 64. Considering 4.2 GHz (boost speed), we get 27.4854/4.2=6.54 Flops/cycle. This is for a mult_3 implementation with (i,k,j) loop ordering. The problem size of 64 can easily fit in the L1 cache (512KB) as problem size of 64 is 64*64*8= 27648 bytes.

Achieved performance for SPMM product is for CSR = 3.22302 GFLOPS/s with a problem size of 64. Considering 4.2 GHz (boost speed), we get 3.22302/4.2=0.7673 Flops/cycle. The problem size of 64 can easily fit in the L1 cache (512KB) as problem size of 64 has NNZ=20224. The space required to store indices in uncompressed form is 20224*8 = 161.792kb (8 bytes is for 2 int row/col indices). The total of roughly 322kb can easily fit in L1 cache as CSR also compresses the row indices.

Let us look at the arithmetic intensity of this mult_3 implementation. This implementation's inner loop has 8 flops and accesses 4 doubles in memory (8*4 = 32 bytes). This gives an arithmetic intensity of 8/32 = 0.125 flops/byte. We check the roofline (docker) model (L1 cache) and see that this corresponds to roughly 30 FLOPS/s which matches closely with the performance of mult_3. Looking at SPMM CSR unoptimized implementation, we see it accesses 3 doubles and 1 int (total of 28 bytes) and performs 2 flops in the inner loop giving an arithmetic intensity of 0.07 flops/byte which gives theoretical performance of ~10GFlops/s. The ratio between SPMM and dense matrix is 10/30 = 0.33 which is higher than actual performance ratio observed for size 64 which is 0.11726.

Array of Structs vs Struct of Arrays (AMATH583)

---

6. How does the performance (in GFLOP/s) for sparse-matrix by vector product of struct of arrays (COOMatrix) compare to that of array of structs (AOSMatrix)? Explain what about the data layout and access pattern would result in better performance.

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T	AOS	AOS^T
64	4096	20224	0.935218	0.939609	3.45063	0.966844	0.957592	3.33561	0.971537	0.957592
128	16384	81408	0.920134	0.924375	3.34316	0.95066	0.964372	3.34316	0.937333	0.937333
256	65536	326656	0.705668	0.733792	1.26579	0.847392	0.836887	1.09474	0.672846	0.688866
512	262144	1308672	0.722026	0.781297	1.22449	0.844305	0.802251	1.24635	0.662619	0.673272
1024	1048576	5238784	0.710867	0.710867	0.662044	0.424268	0.613191	1.16417	0.575141	0.584913
2048	4194304	20963328	0.753738	0.750365	1.14086	0.722873	0.729159	1.1606	0.450824	0.462002

Table 7: Sparse-matrix vector multiplication (unoptimized)

As shown in Table 7, COO and AOS SPMV perform almost similarly for all problem sizes except for larger problem sizes e.g. 2048 COO is clearly better but that is not enough to generalize. The layouts do not clearly differ in performance over multiple runs indicating that for this problem there is no performance difference observed. AOS is generally beneficial if we are going to process slices (or tuples) together so they can fit into one cache line and COO is more beneficial if we process single vectors like row_indices, col_indices_ or storage_ in parallel. The problem accesses across row_indices_, col_indices and storage_ slices so one would expect AOS to be slightly better but based on the observed numbers that is not a clear conclusion. It's possible that the performance benefit of AOS over COO is lost as the slices are later used to index into y and x which may not be a contiguous access.

7. How does the performance (in GFLOP/s) for sparse-matrix by dense matrix product of struct of arrays (COOMatrix) compare to that of array of structs (AOSMatrix)? Explain what about the data layout and access pattern would result in better performance.

Given below is a sample run of sparse-matrix by dense matrix product

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC	AOS
64	4096	20224	1	0.667392	3.22302	0.674202	0.674202
128	16384	81408	1	0.649668	3.08214	0.665991	0.637174
256	65536	326656	1	0.575138	1.24613	0.628889	0.480651
512	262144	1308672	1	0.581632	1.11289	0.572213	0.362401
1024	1048576	5238784	1	0.343527	0.725719	0.38187	0.546062
2048	4194304	20963328	1	0.606377	1.23314	0.440671	0.405929

Table 8: Sparse-matrix dense-matrix multiplication (unoptimized)

COO is almost as efficient as AOS implementation overall across multiple runs in some runs COO is more efficient but in other runs AOS is more efficient and there is no clear pattern of one being better than the other. The explanation is same as the previous question: AOS is generally beneficial if we are going to process slices (or tuples) together so they can fit into one cache line and COO is more beneficial if we process single vectors like row_indices, col_indices_ or storage_ in parallel. The problem accesses across row_indices_, col_indices and storage_ slices so one would expect AOS to be slightly better but based on the observed numbers that is not a clear conclusion. It's possible that the performance benefit of AOS over COO is lost as the slices are later used to index into y and x which may not be a contiguous access.

Extra credit:

Experiment with some of the optimizations we developed previously in the course for matrix-matrix product and apply them to sparse-matrix by dense-matrix product. Explain, and quantify if you can, using the roofline model.

Given below is a sample run for SPMM both unoptimized and optimized versions.

unoptimized:

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC	AOS
64	4096	20224	1	0.667392	3.22302	0.674202	0.674202
128	16384	81408	1	0.649668	3.08214	0.665991	0.637174
256	65536	326656	1	0.575138	1.24613	0.628889	0.480651
512	262144	1308672	1	0.581632	1.11289	0.572213	0.362401
1024	1048576	5238784	1	0.343527	0.725719	0.38187	0.546062
2048	4194304	20963328	1	0.606377	1.23314	0.440671	0.405929

Optimized:

N(Grid)	N(Matrix)	NNZ	NRHS	COO	CSR	CSC	AOS
64	4096	20224	1	1.13917	3.14628	0.611776	1.21233
128	16384	81408	1	1.09531	3.08214	0.616429	1.14252
256	65536	326656	1	0.735422	1.29406	0.553837	0.575138
512	262144	1308672	1	0.703167	0.968059	0.581632	0.699686
1024	1048576	5238784	1	0.842418	1.25105	0.539039	0.665242
2048	4194304	20963328	1	0.826723	0.998254	0.441333	0.535559

Hoisting and unrolling optimizations are implemented for COO, CSR, CSC and AOS implementations. Hoisting reduces memory accesses by moving them outside the inner loop. This is beneficial as memory accesses are very slow. Unrolling reduces number of branch instructions to be taken and improves data reuse as some data is cached and reused.

There are performance improvements observed in COO and AOS implementations after optimization. For problem size 64, COO gives 1.13917 GFlops/s compared to 0.667392 GFlops/s for unoptimized version. Arithmetic intensity of optimized COO is 8/64 = 0.125 flops/byte (8 doubles and 8 flops) which is higher than that of unoptimized version which is 2/32 = 0.0625 flops/byte (8 doubles, 2 ints and 2 flops). Considering that problem size 64 would fit in L1 cache, we get around ~15 GFlops/s for optimized and around ~8 GFlops/s for unoptimized giving a theoretical performance ratio of 8/15 = 0.533 indicating that the high arithmetic intensity of optimized version is indeed beneficial.

Experiment with some of the optimizations we developed previously in the course for matrix-matrix product and apply them to sparse-matrix by vector product. Explain, and quantify if you can, using the roofline model.

Given below is a sample run for SPMV both unoptimized and optimized versions.

Unoptimized:

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T	AOS	AOS^T
64	4096	20224	0.935218	0.939609	3.45063	0.966844	0.957592	3.33561	0.971537	0.957592
128	16384	81408	0.920134	0.924375	3.34316	0.95066	0.964372	3.34316	0.937333	0.937333
256	65536	326656	0.705668	0.733792	1.26579	0.847392	0.836887	1.09474	0.672846	0.688866
512	262144	1308672	0.722026	0.781297	1.22449	0.844305	0.802251	1.24635	0.662619	0.673272
1024	1048576	5238784	0.710867	0.710867	0.662044	0.424268	0.613191	1.16417	0.575141	0.584913
2048	4194304	20963328	0.753738	0.750365	1.14086	0.722873	0.729159	1.1606	0.450824	0.462002

Optimized:

N(Grid)	N(Matrix)	NNZ	COO	COO^T	CSR	CSR^T	CSC	CSC^T	AOS	AOS^T
64	4096	20224	0.962196	0.944041	4.00273	0.985895	0.985895	3.77616	0.995705	0.957592
128	16384	81408	0.891508	0.91177	3.85749	0.964372	0.978484	3.85749	0.95066	0.928654
256	65536	326656	0.686531	0.715642	0.892188	0.688866	0.775964	1.00261	0.627018	0.559466
512	262144	1308672	0.654336	0.790142	1.24635	0.865238	0.814737	1.29252	0.697958	0.69564
1024	1048576	5238784	0.757812	0.743778	1.11053	0.857595	0.851534	1.10038	0.698505	0.698505
2048	4194304	20963328	0.806282	0.760574	1.0226	0.866701	0.853469	1.25623	0.200366	0.565621

There are some improvements in COO and AOS performance. Following optimizations are implemented:

1. COO: Hoisting, unrolling, SIMD
2. CSR: hoisting, unrolling
3. CSR^T: hoisting.
4. CSC: hoisting.
5. CSC^T: hoisting.
6. AOS: Hoisting, unrolling, SIMD

Hoisting reduces memory accesses by moving them outside the inner loop. This is beneficial as memory accesses are very slow. Unrolling reduces number of branch instructions to be taken and improves data reuse as some data is cached and reused. SIMD enables processing 2 doubles in parallel.

There is some performance improvement observed for CSR and CSC^T for smaller problem sizes (128 and below).

For problem size 64, CSR gives 4.00273 GFlops/s for optimized compared to 3.45063 GFflops/s for unoptimized. Arithmetic intensity for optimized CSR is 4/40 = 0.1 flops/byte (4 doubles, 2 int, 4 flops) which is higher than that of unoptimized version which is 2/28 = 0.07 flops/byte (3 doubles, 1 int, 2 flops). Considering that problem size 64 would fit in L1 cache, we get around ~14 GFlops/s for optimized and around ~8 GFlops/s for unoptimized giving a theoretical performance ratio of 8/14 = 0.5714 indicating that the high arithmetic intensity of optimized version is indeed beneficial.

About Midterm

The most important thing I learned from this assignment was ...

How can we store sparse matrices in an optimized way and what are the different ways of doing that.

One thing I am still not clear on is ...

Why is there little difference between COO and AOS performance