PARALLEL COMPUTING: MATRIX MULTIPLICATION

Suraj Anand
Brown University
suraj_anand@brown.edu

1 Matrix-Vector Multiplication

I wrote the code for matrix-vector multiplication as a nested for-loop that iterates through rows of the matrix and sums up entries of the vector multiplied by entries of the matrix-row in OSCAR.

This code originally uses non-contiguous memory in the random initialization of the matrix without compiler optimizations.

For a matrix of N rows and M columns by a vector of M elements, I varied M and N from $O(10)$ to $O(10000)$ and calculated the time taken to perform the matrix-vector multiplication.

N (# Rows)	M (# Cols)	milliseconds	floprate
10	10	0	∞
	100	3	666.667
	1000	27	740.741
	10000	266	751.88
100	10	3	666.667
	100	27	740.741
	1000	268	746.269
	10000	2662	751.315
1000	10	28	714.286
	100	275	727.273
	1000	2556	782.473
	10000	27719	721.527
10000	10	314	636.943
	100	2871	696.621
	1000	27914	716.486
	10000	296700	674.082

Table 1: Time and Floprate of Matrix Multiplication

The average floprate was 715.598, excluding the floprate that was calculated at ∞ which occurred due to rounding error.

2 Roofline Model

For each operation in the for-loop, one addition and one multiplication operation occur. Additionally, for each operation in the for-loop, three loads and one store occur. Thus, given a matrix of size $N \times M$ and a vector of size M, we can calculate that there are $2MN$ FLOPS and there are $32MN$ Bytes. As such, the arithmetic intensity of this matrix multiplication is $\frac{1}{16}$ which lies on the positive sloped piece of the roofline model.

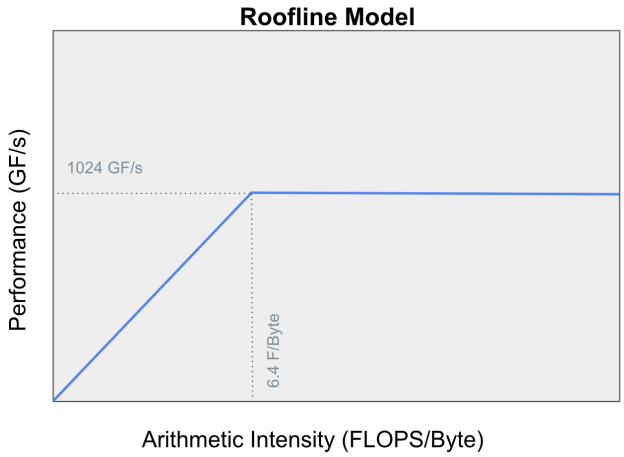

Figure 1: Naive Roofline Model (peak bandwidth of 160 GB/s, peak flop rate of 1 TFLOP/s)

3 Optimizations

3.1 Contiguous Memory

I implemented this by allocating an array of memory NM for the matrix and then manually rows of the array to point at the memory addresses offset by the number of rows before times the number of columns.

N (# Rows)	M (# Cols)	milliseconds	floprate
10	10	0	∞
	100	3	666.667
	1000	27	740.741
	10000	279	716.846
100	10	3	666.667
	100	28	714.286
	1000	270	740.741
	10000	2758	725.163
1000	10	28	714.286
	100	275	727.273
	1000	2667	749.906
	10000	27936	715.922
10000	10	301	664.452
	100	2912	686.813
	1000	27941	715.794
	10000	275466	726.042

Table 2: Time and Floprate of Contiguous Matrix Multiplication

This technique actually decreased the floprate by a small (likely not statistically significant) amount, which is counter to what I thought would happen. On average, the floprate was 666.975 not including the entry that was ∞.

3.2 Compiler Optimizations

I used compiler optimizations simply by calling $-O1$ for little optimization, $-O2$ for moderate optimization, and $-O3$ for extreme optimization. The floprate with each of these optimizations can be seen below in the table. Note, these optimizations are all done with contiguous matrix memory.

N (# Rows)	M (# Cols)	-O1 floprate	-O2 floprate	-O3 floprate
10	10	∞	∞	∞
	100	1000	2000	2000
	1000	769.231	1666.67	1666.67
	10000	772.201	1626.02	1724.14
100	10	2000	2000	2000
	100	869.565	2222.22	2222.22
	1000	781.25	1754.39	1769.91
	10000	765.697	1690.62	1611.6
1000	10	1666.67	2857.14	3333.33
	100	847.458	1851.85	2000
	1000	768.049	1706.48	1646.09
	10000	776.88	1552.07	1532.92
10000	10	1724.14	2469.14	2816.9
	100	810.701	1638	1655.63
	1000	734.322	1575.67	1655.63
	10000	776.654	1539.84	1344.96

Table 3: Floprate of Matrix Multiplication with Compiler Optimizations

The mean floprate for $-O1$ was 941.426 FLOPS, the mean floprate for $-O2$ was 1759.38 FLOPS, and the mean floprate for $-O3$ was 1806.56 FLOPS. One interesting note here is that the FLOPS increased by a factor of two when increasing optimization amount from $-O1$ to $-O2$, but increased only by 40 FLOPS when changing from $-O2$ to $-O3$.

3.3 Loop Unrolling

I implemented loop unrolling by a factor of 2 as well as a factor of 4. I did this by incrementing the column index by either 2 or 4. If the number of columns was not divisible by 2 or 4, I would add the remaining elements at the end. The table below shows the floprate with loop unrolling. Note that this is with no compiler optimizations, but is with contiguous matrix memory.

N (# Rows)	M (# Cols)	unroll 2 floprate	unroll 4 floprate
10	10	∞	∞
	100	1000	1000
	1000	833	1000
	10000	836	985
100	10	1000	1000
	100	909	1111
	1000	934	1129
	10000	937	1111
1000	10	909	1176
	100	917	1123
	1000	940	1112
	10000	908	1080
10000	10	865	1104
	100	838	1064
	1000	909	1075
	10000	900	509

Table 4: Floprate of Matrix Multiplication with Unrolling

The average floprate with unrolling two operations was 864.688 while the average floprate with unrolling four operations was 986.188. The disadvantage of unrolling is that the code becomes more unreadable and less elegant.

4 Conclusion

I was fairly surprised with how much the compiler optimizations helped increase FLOP rate. I am sure that multithreading with OpenMP could further increase the floprate of matrix multiplication, potentially getting it closer to the upper bound indicated by the roofline model. It was also interesting that the floprate did not change very much depending on the size of the matrix and vector, which shows the validity of arithmethic intensity and the roof line model as a method of estimating CPU use efficiency.