Name: Carlos Moustafa
Net ID: aem278
Perlmutter Username: aem278

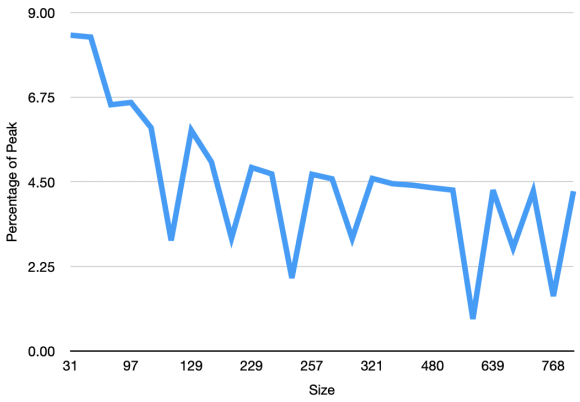
Optimizations

Defaults

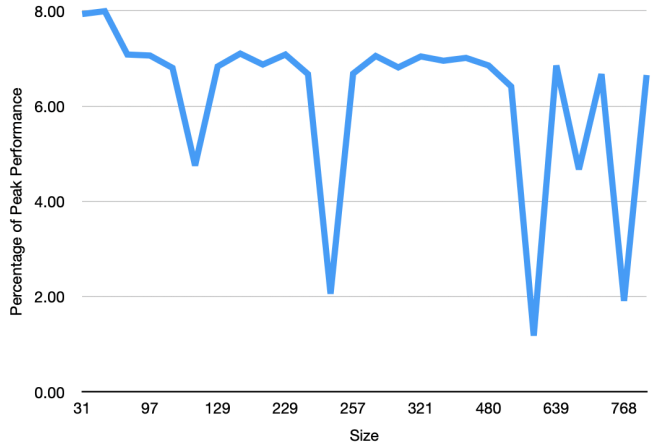
Below, I give figures for the performance of the default naive implementation and the default blocked implementation as a baseline comparison for my optimizations.

Default Naive

The default naive implementation gave an average peak performance of 4.48%, backed by the data and figure above.



Default Blocked



The default blocked implementation (with block size 41) gave an average peak performance of 6.19%, backed by the data and figure above. Interestingly, the figure seems to have fewer relative dips than the naive implementation, which I expect relates to the size of the cache and cache misses.

Configuring Block Size

Initially, I targeted the L1 cache of Perlmutter for configuring my block size. I observed that Perlmutter has an L1 data-only cache size of 31752 bytes. From this, I first chose a block size of 63, since each entry of the matrix should be 8 bytes, thus an entire 63x63 matrix of doubles should have a size of 31752 bytes, which should entirely fit in the L1 cache.

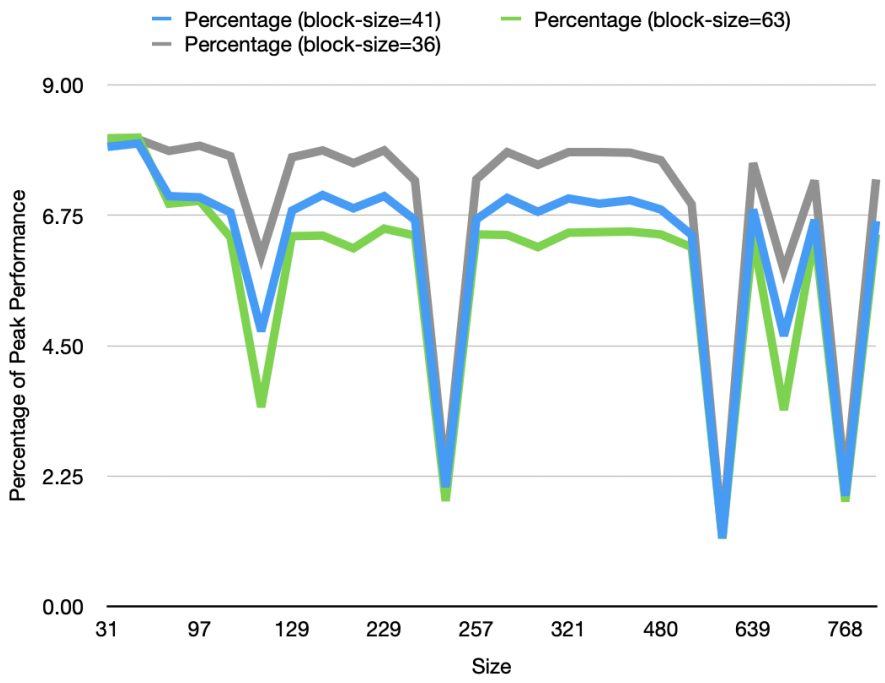
However, this choice of block size gave a worse performance than the default block size, and I realized this is most likely because I need 3 blocks to fit in the target cache at once (1 from each matrix). With this in mind, I computed a block size of 36 for fitting 3 blocks in the L1 cache, and

the results below support this was an improved choice.

The 63-block size and 36 block-size configurations gave average performances of 5.78% and 6.88% of peak, respectively.

In hindsight, after reading Anatomy of High-Performance Matrix Multiplication, I would have focused on targetted the L2 cache for the largest-level block, since the difference in time-to-registers between L1 and L2 is small relative to the spatial increase the L2 cache offers.

Size	Mflops/s (block-size=41)	Percentage (block-size=41)	Mflops/s (block-size=63)	Percentage (block-size=63)	Mflops/s (block-size=36)	Percentage (block-size=36)
31	4441.16	7.93	4524.25	8.08	4516.24	8.06
32	4472.80	7.99	4528.37	8.09	4512.43	8.06
96	3962.78	7.08	3889.81	6.95	4404.38	7.86
97	3952.34	7.06	3920.53	7.00	4452.38	7.95
127	3805.58	6.80	3569.90	6.37	4353.46	7.77
128	2657.99	4.75	1926.01	3.44	3390.21	6.05
129	3823.88	6.83	3578.24	6.39	4340.91	7.75
191	3973.42	7.10	3585.19	6.40	4405.98	7.87
192	3845.59	6.87	3461.94	6.18	4285.58	7.65
229	3966.99	7.08	3651.46	6.52	4405.43	7.87
255	3737.37	6.67	3585.30	6.40	4116.96	7.35
256	1156.00	2.06	1019.76	1.82	1253.36	2.24
257	3741.27	6.68	3595.32	6.42	4128.76	7.37
319	3946.12	7.05	3592.07	6.41	4389.90	7.84
320	3814.10	6.81	3470.58	6.20	4264.99	7.62
321	3940.91	7.04	3613.18	6.45	4391.55	7.84
417	3894.25	6.95	3617.15	6.46	4388.17	7.84
479	3923.31	7.01	3623.87	6.47	4383.80	7.83
480	3833.42	6.85	3596.75	6.42	4313.34	7.70
511	3589.82	6.41	3471.38	6.20	3885.20	6.94
512	658.20	1.18	654.82	1.17	694.58	1.24
639	3834.66	6.85	3581.12	6.39	4281.85	7.65
640	2617.04	4.67	1898.95	3.39	3250.13	5.80
767	3734.80	6.67	3593.46	6.42	4116.17	7.35
768	1071.90	1.91	1011.99	1.81	1185.14	2.12
769	3723.59	6.65	3594.42	6.42	4124.53	7.37



Interestingly, the plots for each block size have similar shapes, and they all seem to coincide at matrix sizes 256, 512, and 768 with their poorest performance.

AVX Intrinsics

My next choice of optimization was to introduce AVX Intrinsics for utilizing hardware-level concurrency for computing the dot products between rows of A and columns of B.

One thing I learned from this is that AVX seems to require any loaded data be contiguous in memory, so this led me to transpose the matrix A to be column-major. In particular, I decided to copy A to a separate memory-aligned location using _m_malloc in transposed format. I also copied B memory-aligned using the same method, to avoid cache misses due to the data not being word-aligned.

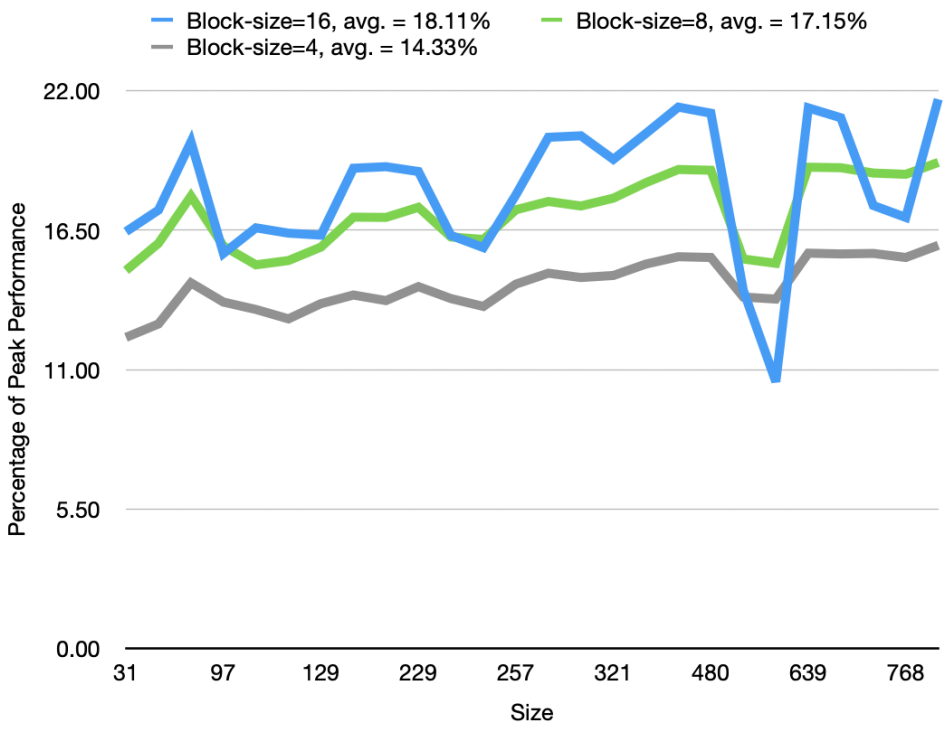
One issue I ran into with the AVX functions I used was that they load 4 numbers at a time into a 256-byte vector. Therefore, when K (as defined in the slides) was not a multiple of 4, it would load irrelevant memory, possibly causing a segmentation fault. To get around this, I decided to pad matrices A and B with zeroes so that they are a multiple of the smallest block size, and I only chose block sizes that are a multiple of 4.

In hindsight, I realized I could have just checked if K is less than the smallest block size, and use non-AVX dot product calculations in those cases. This would have avoided unnecessary padding.

Overall, switching to AVX to compute the dot product yielded the single-highest increase in performance out of all optimizations I attempted in this project.

The data and figure below is for blocked matrix multiplication combined with AVX for computing dot products, as well as the transposing of A and padding described above. This set of optimizations, with the block size of 16, ended up being my highest-performance optimization set, and is the configuration of the code I submitted.

Size	Mflops/s (block-size=16)	Percentage (block-size=16)	Mflops/s (block-size=8)	Percentage (block-size=8)	Mflops/s (block-size=4)	Percentage (block-size=4)
31	9209.03	16.44	8350.98	14.91	6870.85	12.27
32	9683.57	17.29	8947.32	15.98	7165.23	12.80
96	11188.87	19.98	9989.24	17.84	8074.09	14.42
97	8724.12	15.58	8885.31	15.87	7649.06	13.66
127	9288.17	16.59	8473.09	15.13	7485.93	13.37
128	9174.13	16.38	8570.79	15.30	7280.79	13.00
129	9125.98	16.30	8866.38	15.83	7614.45	13.60
191	10606.50	18.94	9526.39	17.01	7808.24	13.94
192	10639.33	19.00	9520.92	17.00	7683.02	13.72
229	10532.57	18.81	9743.23	17.40	7988.73	14.27
255	9122.54	16.29	9086.38	16.23	7725.46	13.80
256	8853.36	15.81	9031.67	16.13	7552.74	13.49
257	10017.24	17.89	9689.82	17.30	8049.17	14.37
319	11291.38	20.16	9875.09	17.63	8286.50	14.80
320	11323.80	20.22	9769.40	17.45	8193.92	14.63
321	10800.07	19.29	9944.05	17.76	8238.58	14.71
417	11373.66	20.31	10286.87	18.37	8492.11	15.16
479	11957.60	21.35	10577.92	18.89	8649.78	15.45
480	11824.16	21.11	10561.15	18.86	8632.77	15.42
511	7908.20	14.12	8602.94	15.36	7761.18	13.86
512	5891.16	10.52	8507.29	15.19	7717.92	13.78
639	11940.60	21.32	10627.19	18.98	8732.50	15.59
640	11723.38	20.93	10617.34	18.96	8715.46	15.56
767	9783.48	17.47	10500.95	18.75	8722.05	15.58
768	9521.41	17.00	10470.01	18.70	8637.52	15.42
769	12129.14	21.66	10735.24	19.17	8902.36	15.90



The figure above seems to imply that with AVX for dot products, the larger the block size, the better the performance, barring a notable exception at matrix size 512, where the 16-block size performed worst. This intrigues me and I don't have any possible explanations.

If I had more time, I would have experimented with even higher values for the AVX kernel block size, to see at what point increasing block size starts giving diminishing returns on performance.

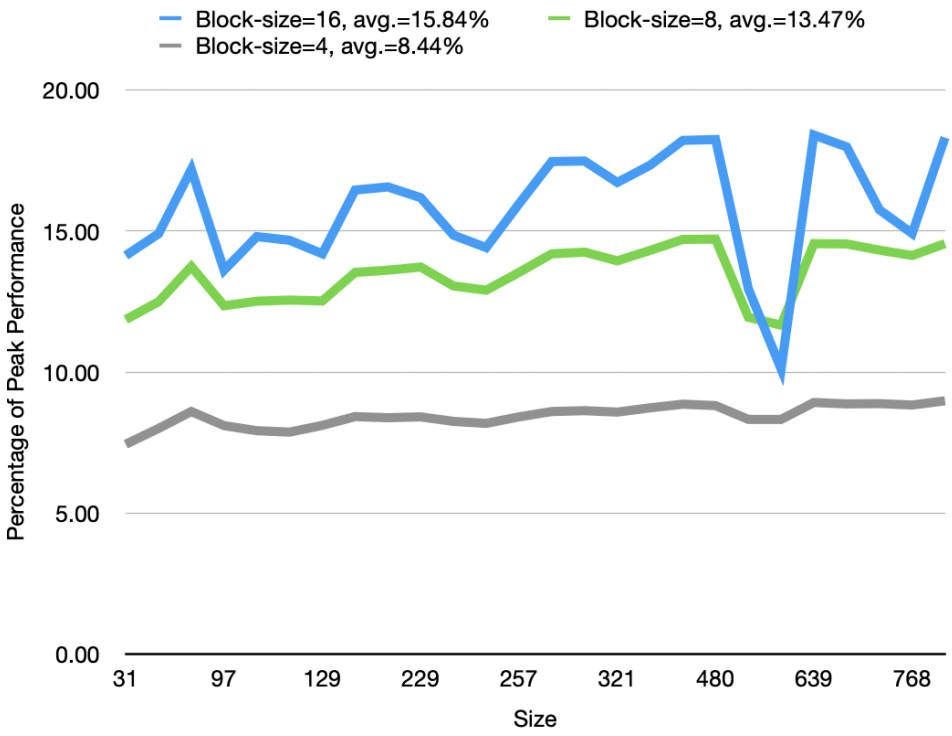
Repacking

The next optimization I experimented with is to repack A so that it was ordered block-contiguous in memory. I was able to achieve this "on-the-fly", meaning, I was able to repack the matrix during the iterations over the blocks. However, this led to decreased performance across-the-board, and I suspect this is because of the fact that I was introducing significant memory operations interleaved with my compute code.

I tried to repack the matrices to be block-contiguous in a pre-process, but could not figure out how to get the algorithm working with it. In hindsight, I should have sought out help early for this, but I expect this would have given a significant performance increase if done correctly.

The data and figure below is for blocked matrix multiplication combined with AVX for computing dot products, transposing A, padding both matrices, and copying blocks of A to a separate memory location on-the-fly.

Size	Mflops/s (block-size=16)	Percentage (block-size=16)	Mflops/s (block-size=8)	Percentage (block-size=8)	Mflops/s (block-size=4)	Percentage (block-size=4)
31	7913.38	14.13	6643.69	11.86	4167.82	7.44
32	8344.36	14.90	6996.91	12.49	4480.53	8.00
96	9619.77	17.18	7693.82	13.74	4813.80	8.60
97	7621.22	13.61	6915.54	12.35	4533.46	8.10
127	8289.17	14.80	7003.23	12.51	4433.26	7.92
128	8213.46	14.67	7030.15	12.55	4408.96	7.87
129	7948.30	14.19	7013.12	12.52	4543.75	8.11
191	9209.97	16.45	7578.59	13.53	4713.42	8.42
192	9275.94	16.56	7620.43	13.61	4691.88	8.38
229	9063.18	16.18	7685.20	13.72	4707.55	8.41
255	8318.16	14.85	7307.33	13.05	4621.07	8.25
256	8069.64	14.41	7221.86	12.90	4583.15	8.18
257	8937.36	15.96	7576.89	13.53	4711.45	8.41
319	9779.78	17.46	7946.95	14.19	4813.58	8.60
320	9786.49	17.48	7979.56	14.25	4835.18	8.63
321	9365.26	16.72	7805.42	13.94	4802.32	8.58
417	9702.19	17.33	8011.05	14.31	4887.50	8.73
479	10196.40	18.21	8233.41	14.70	4963.36	8.86
480	10216.95	18.24	8239.15	14.71	4931.19	8.81
511	7252.53	12.95	6688.69	11.94	4660.75	8.32
512	5659.19	10.11	6527.96	11.66	4661.13	8.32
639	10301.41	18.40	8150.73	14.55	4993.67	8.92
640	10071.06	17.98	8139.89	14.54	4965.89	8.87
767	8821.99	15.75	8017.26	14.32	4972.22	8.88
768	8349.61	14.91	7911.21	14.13	4947.33	8.83
769	10252.40	18.31	8149.46	14.55	5026.33	8.98



Multi-Level Blocking

Next, I attempted to implement two-level blocking to further optimize the multiplication. Taking inspiration from Anatomy of High-Performance Matrix Multiplication, I wanted to target L2 for storing the larger-level block of B, L1/L2 for storing the larger-level block of A, and L1/registers for storing the smaller-level blocks of all three matrices.

In general, introducing a second level did not significantly increase performance–it slightly decreased it. I believe this is because I did not copy the higher-level blocks to a new memory location and/or repack the matrices to be block-contiguous in a pre-process. If I had more time, I would've focused more time on this and expect that it would've given a significant performance increase, especially in combination with repacking.

The data and figure below is for two-level blocked matrix multiplication combined with AVX for computing dot products, transposing A, and padding both matrices.

Size	Mflops/s (blocks=32&16)	Percentage (blocks=32&16)	Mflops/s (blocks=32&8)	Percentage (blocks=32&8)	Mflops/s (blocks=32&4)	Percentage (blocks=32&4)	Mflops/s (blocks=16&8)	Percentage (blocks=16&8)
31	9195.08	16.42	8342.83	14.90	7216.24	12.89	8258.79	14.75

32	9673.86	17.27	8720.07	15.57	7560.66	13.50	8586.22	15.33
96	10756.41	19.21	9774.17	17.45	8293.26	14.81	9648.82	17.23
97	6449.59	11.52	6269.77	11.20	5589.62	9.98	8206.69	14.65
127	9244.84	16.51	8551.63	15.27	7377.54	13.17	8450.44	15.09
128	9220.82	16.47	8683.20	15.51	7468.84	13.34	8623.94	15.40
129	8274.50	14.78	7955.12	14.21	6983.59	12.47	8437.01	15.07
191	10418.22	18.60	9528.22	17.01	8145.45	14.55	9442.68	16.86
192	10490.73	18.73	9588.53	17.12	8189.53	14.62	9506.74	16.98
229	8134.19	14.53	8427.15	15.05	7267.09	12.98	9449.91	16.87
255	9043.87	16.15	9153.00	16.34	7788.48	13.91	8754.60	15.63
256	8785.30	15.69	9131.23	16.31	7773.32	13.88	8868.10	15.84
257	10046.52	17.94	9349.64	16.70	8036.76	14.35	9332.70	16.67
319	11161.10	19.93	10021.71	17.90	8581.39	15.32	9971.23	17.81
320	11199.62	20.00	10194.19	18.20	8604.20	15.36	9984.55	17.83
321	10587.18	18.91	9823.84	17.54	8356.13	14.92	9700.77	17.32
417	10956.51	19.57	10135.01	18.10	8535.17	15.24	10073.87	17.99
479	11700.96	20.89	10629.76	18.98	8899.20	15.89	10434.46	18.63
480	11713.18	20.92	10617.15	18.96	8902.84	15.90	10442.56	18.65
511	7858.56	14.03	8777.13	15.67	7505.89	13.40	8624.17	15.40
512	5876.68	10.49	8571.97	15.31	7048.25	12.59	8452.90	15.09
639	11675.78	20.85	10595.74	18.92	8783.46	15.68	10463.54	18.68
640	11398.97	20.36	10606.06	18.94	8812.01	15.74	10439.40	18.64
767	9650.75	17.23	10444.23	18.65	8729.14	15.59	10160.49	18.14
768	9364.02	16.72	10392.65	18.56	8573.27	15.31	10126.20	18.08

769	11838.88	21.14	10939.62	19.54	9078.69	16.21	10625.23	18.97