CS141

Elijah Tai

May 7, 2019

Tiled Matrix Multiplication

Why Matrix Multiplication is Important

Matrix multiplication is extremely important for Machine Learning. The three main reasons are:

1. Data often comes in the form of matrices
2. Matrix operations are common in Deep Learning
3. Collected data can be described as matrices[1]

One reason is because often data comes in the form of matrices. For example, in the area of image recognition, images can be stored as matrices of values of pixels, and so linear manipulation of the images can be thought of as matrix multiplications[2][3]. Therefore, when working in such areas, matrix operations including multiplication is extremely common. Another reason is due to the recent developments in Deep Learning, which is highly reliant on matrices. Weights and bases stored in a neural network are just matrices of values, and as initial inputs progress layer by layer, they are being weighted and processed through some activation function. As the networked is trained, inputs are fed in and evaluated via some cost function, and the matrices are adjusted in accordance to some gradient descent[4]. Such operations are heavily dependent on matrix multiplication. As such, matrix multiplication is extremely important when working with these types of networks. (Also, matrix multiplication is important when dealing with graphs in general). Finally, many text-based Machine Learning optimizations require matrix multiplication. For example, in the Bag of Words model which is commonly used for document type classification, a matrix of frequencies is used to train a classifier. Term-document matrices store frequencies of words and are important for natural language processing and semantic analysis[5]. These are examples of where information (in this case frequency) from data can be generalized and stored as matrices. For these reasons, matrix multiplication is common in ML, and optimizing this process can lead to huge improvements in runtime of algorithms.

Current Areas of Research on Matrix Multiplication Optimizations

Because matrix multiplication is so important to ML and computer science in general, ever since Strassen, researchers have been trying to improve runtime. For example, when analyzing data such as Facebook friendships as a graph where friendships are edges, the adjacency list representation is very

---

[1] Yadav, Vikas kumar. “A Comprehensive Beginners Guide to Linear Algebra for Data Scientists.” Analytics Vidhya, 12 Nov. 2018.

[2] Ma, Zhigang, et al. Thinking of Images as What They Are: Compound Matrix Regression for Image Classification. *Twenty-Third International Joint Conference on Artificial Intelligence*.

[3] Formont, Pierre, et al. “On the Use of Matrix Information Geometry for Polarimetric SAR Image Classification.” Matrix Information Geometry, 2012, pp. 257–276., doi:10.1007/978-3-642-30232-9_10.

[4] Ross, Matt. “Under The Hood of Neural Network Forward Propagation - The Dreaded Matrix Multiplication.” Towards Data Science, Towards Data Science, 10 Sept. 2017.

[5] D'Souza, Jocelyn. “An Introduction to Bag-of-Words in NLP.” Medium, GreyAtom, 3 Apr. 2018.

sparse. In particular, the average number of friendships is 338 out 1.08 billion users, meaning that friendship density is only 0.0003%. Thus, researchers look for ways to speed up sparse matrix multiplication[6]. Another area of optimization is parallelization of matrix multiplication (in a multiprocessor system) and making such divisions scalable[7]. These advancements help speed up matrix multiplication. However, there are also space optimizations. For example, through compressed linear algebra, researchers show how large datasets can be compressed into available memory while minimizing the change in performance compared to the uncompressed matrix[8].

Therefore, optimizing matrix multiplications is important and is something to consider when building AI enabled systems.

Results of Naive Matrix Multiplication and Tiled Matrix Multiplication

Size	8x8	32x32	128x128	256x256	512x512	1024x1024
Naive Avg.	0.00009s	0.000606s	0.0382s	0.337s	3.25s	41.83s
Tiled Avg. 8x8	0.000010s	0.000650s	0.0400s	0.343s	2.78s	21.64s
Tiled Avg. 32x32	0.00009s	0.000602s	0.0406s	0.335s	2.62s	21.34s
Tiled Avg. 128x128	0.00010s	0.000601s	0.0369s	0.297s	2.51s	22.69s
Tiled Avg. 256x256	0.00010s	0.000603s	0.0388s	0.315s	2.64s	26.77s
Tiled Avg. 512x512	0.00009s	0.000592s	0.0393s	0.331s	3.36s	35.64s
Tiled Avg. 1024x1024	0.00010s	0.000618s	0.0389s	0.330s	3.27s	38.66s
Iterations	16384	1024	64	16	4	1

Why the baseline Naive Matrix Multiplication is Insufficient

The reason why the naive triple-for-loop implementation is insufficient is because it requires many slower memory fetches. This is because the cache has limited storage, so for large matrices, it is unlikely to get a cache hit if we calculate row by row and column by column—as we progress through the calculation, we are likely to remove data from cache that we need to access later.

In other words, this is an issue of locality; if we perform calculations on smaller matrices (tiles), then in each calculation, we would be more likely have cache hits and thus improve runtime. So, we can break

[6] Pal, Subhankar, et al. "OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator." 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2018, doi:10.1109/hpca.2018.00067.

[7] Li, Keqin. "Scalable Parallel Matrix Multiplication on Distributed Memory Parallel Computers." Journal of Parallel and Distributed Computing, vol. 61, no. 12, 2001, pp. 1709–1731., doi:10.1006/jpdc.2001.1768.

[8] Elgohary, Ahmed, et al. "Compressed Linear Algebra for Large-Scale Machine Learning." The VLDB Journal, vol. 27, no. 5, 2017, pp. 719–744., doi:10.1007/s00778-017-0478-1.

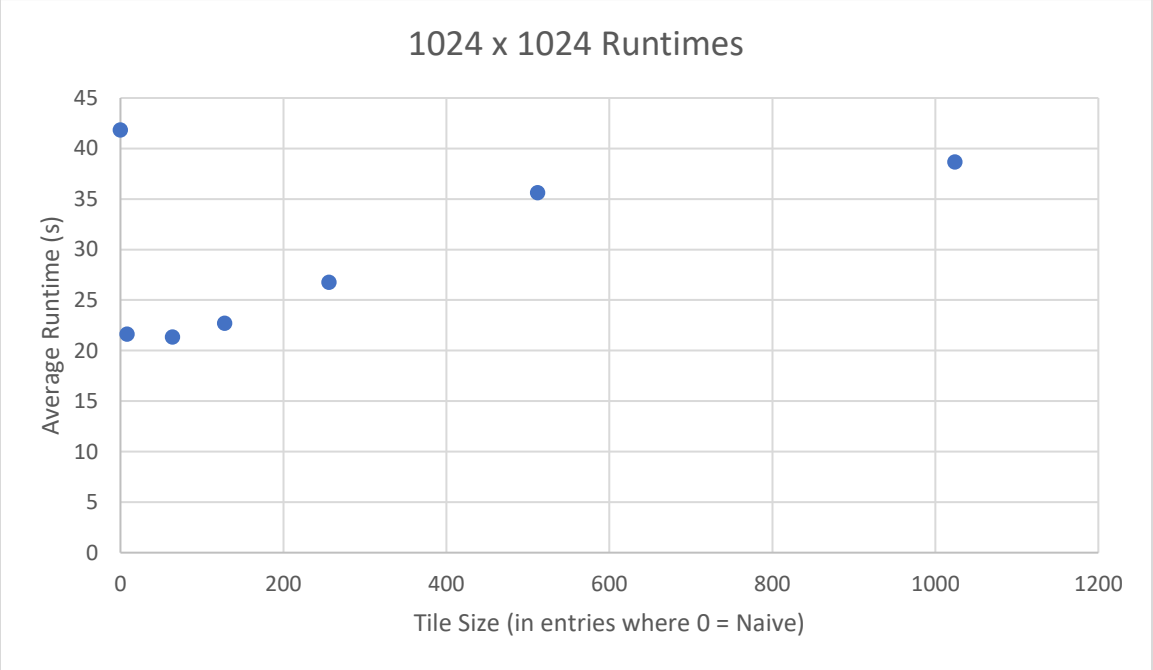
up the original matrix into subproblems with smaller matrices where there will be more cache hits, and then recombine the subproblems to answer to original matrix multiplication.

Tiled Matrix Multiplication

My implementation breaks the original matrix into tiles of a given size. Suppose we are doing A = BC. Then for each row of tiles in B, for each column of tiles in C, my algorithm multiplies the tiles, and increments the values in A by that amount. When all the tiles are done the result is a valid matrix multiplication.

Analysis of Results

Tiling greatly improved runtime for larger matrices such as in the case of the 1024x1024 multiplication, bringing the runtime down from 41.83 seconds to 21.34 seconds



There are a few things I noticed:

1) When the tile size exceeds the matrix size, there is no significant speed difference. My guess is that this is because the tile is so large that it does not divide the matrix into smaller regions, and thus does not improve locality.

2) For smaller matrices, the naive implementation can outperform my implementation. Perhaps this is because small matrices can be stored well in cache, so improving locality does not help. Plus, dividing the matrix into smaller matrices has some overhead, especially when calculating the indexes of entries of the smaller matrices. This might be why my tiled implementation is slower.

3) When tile size is too small, the runtime is slower. Similarly, when tile size is too large, the runtime is slower. Perhaps the former can be explained because there is an overhead of breaking the matrix into a lot of smaller matrices and calculating the indexes. Or, perhaps it is because as the tile size approaches 1, the implementation begins to have the same locality problems as the naive implementation (a tile-

size of 1 is equivalent to the naive implementation). Perhaps the latter can be explained because the tile is so large that the tile itself can have further improvements in locality. Therefore, the best tile size may be an optimization problem based on these factors.

Extra Credit 1: Parameterize Matrix-Multiplication Sizes

I fixed my implementation to deal with matrices of any size. I perform the tiled matrix multiplication for as many tiles as possible, and then separately handle the remaining entries that cannot fit into a whole tile. This means I can choose non-square matrices of arbitrary size. This also means that I can analyze how the shape of matrices affects runtime of the tiled implementation as well as the naive implementation. I arbitrarily chose a tile size of 60 x 60. For simplicity, I made the resulting matrices square, but my implementation should work for non-square resulting matrices as well.

Size	B = 125x2000 C = 2000x125	B = 250x1000 C = 1000x250	B = 500x500 C = 500x500	B = 1000x250 C = 250x1000
Naive Avg.	0.65s	1.29s	2.76s	5.63s
Tiled Avg. 60 x 60	0.62s	1.21s	2.49s	4.94s
Iterations	30	30	30	5

Analysis of Results

If we look at the ratio of tilted times to naive times, we see that:

Size	B = 125x2000 C = 2000x125	B = 250x1000 C = 1000x250	B = 500x500 C = 500x500	B = 1000x250 C = 250x1000
Tiled / Naive	0.95	0.94	0.90	0.87

It seems as if shape does an impact. Specifically, for matrices with the same number of elements, matrices with larger inner dimensions seem to benefit more from tiled matrix multiplication.

Extra Credit 2: Different Integer Sizes int32 int16 int8

Type	Integer Size	Matrix Size	Avg. Time	Iterations
Naive	int32	512x512	3.78s	5
	int16	512x512	3.12s	5
	int8	512x512	3.06s	5
Tiled	int32	512x512	2.66s	5
	int16	512x512	2.97s	5
	int8	512x512	3.02s	5

It seems as if the precision does affect runtime, especially in the naive case, greatly decreasing the average time per matrix multiplication, because using smaller data types decreases the size of the

matrix allowing it to be stored in cache, allowing it's runtime to be very similar to the tiled implementation (at int8, the smallest data type tested).

Extra Credit 3: Strassen's Algorithm

Currently, we have been optimizing based on hardware and cache hits. However, there are also optimizations on the software side. So, as an additional exploration into matrix multiplication, I implemented Strassen's algorithm, the first matrix multiplication algorithm to break O(n^3). Strassen runs in O(n^2.8), because instead of having to do 8 recurrences, it cleverly finds a way to omit one recurrence. Thus, the recurrence is T(n) = 7T(n/2) + O(n^2). I implemented this in Python.

Conclusion

Matrix multiplication is extremely important for ML. There are many ways to do better than naively multiplying out the rows and the columns. In terms of hardware, we can encourage cache hits by tiling, and using lower precision data types. Perhaps, certain shapes may be faster as well. On the software side, there is parallelization, compressed linear algebra, and algorithms like Strassen's algorithm to improver runtime. Such algorithms and techniques will likely to become more and more important given advances in ML and the need for faster multiplication.