CACHING AND ITS EFFECTS ON TILED MATRIX MULTIPLICATION

Victor Qin

CS 141 Final Project - May 7, 2019

Current systems for machine learning systems commonly involve large matrix multiplications with thousands of indices and up to trillions of calculations. When combined with the massive data sets necessary for machine learning, any sort of optimization for the multiplications between matrices will have significant time savings for algorithms in machine learning.

One simple and fundamental optimization is to tile matrices when multiplying them. This has advantages from a major time-intensive step in the multiplication process, which is actually fetching the data that must be multiplied. First fetching the value for each index of the matrix causes delays if the value isn't in a local cache, because the processor must reach into the main memory that is much farther away rather than being able access it in the physically closer and quicker data access in the cache. Matrix multiplication is especially prone to this error, because its repetitive use of the same values over and over requires many slow fetches to the main memory if the matrices being multiplied are larger than the space available in the cache and forcing the cache to drop some values between cycles of multiplication.

Tiling minimizes the number of slow fetches to main memory by maximizing the use of a value while it's in the cache. By dividing the matrices being multiplied into smaller blocks that can fit into the cache, the algorithm minimizes the number of slow fetches required. We will demonstrate the improvements tiling can bring to matrix multiplication processes by comparing a naïve vs. a tiled matrix multiplication algorithm implemented in C and their performance on matrices of different sizes and then analyze how some of these changes may be traced back to the caches of the processor.

What is Tiling

The naïve implementation of the matrix multiplication simply iterates along the output matrix, by fully calculating each index of the output matrix before moving on to the next index. This gives the output for that index of the matrix immediately, but is very inefficient in using the cache. For example, given the 4x4 matrix multiplication:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \times \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

which we can also represent as:

$$A = BC$$

we can see that indices of the output matrix rely on some of the same inputs. For example, all four of the output indicesa_{11} thru $a14$ depend on b_{11} thru b_{14}, but on different columns of C. By leveraging the repetitive structure that is contained in matrix multiplication, we can improve the reuse of some of these values. Let us do the matrix multiplication above, this time with tiling. We first divide the matrices into smaller square matrices that will fit in our cache memory - for example, on the A matrix:

$$\left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right] = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

Thus,

$$A_{11} = B_{11} \times C_{11} + B_{12} \times C_{21}$$

If we execute this equation, the indices of A_{11} Will first be partially calculated through all of the values in B_{11} and C_{11}. This has significant time saving effects on our matrix multiplication algorithm.

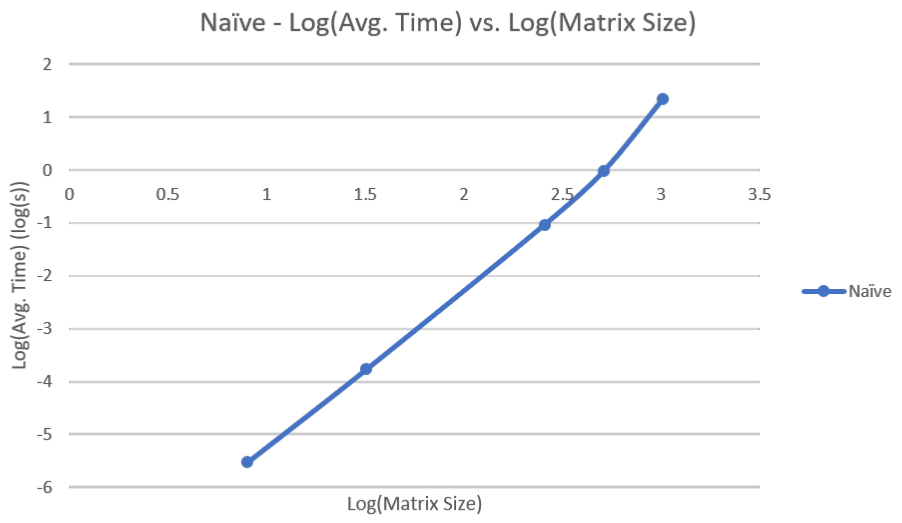
Naive Implementation Results

Below are several graphs and tables illustrating the performance of the naive matrix multiplication algorithm.

Execution Time Data:

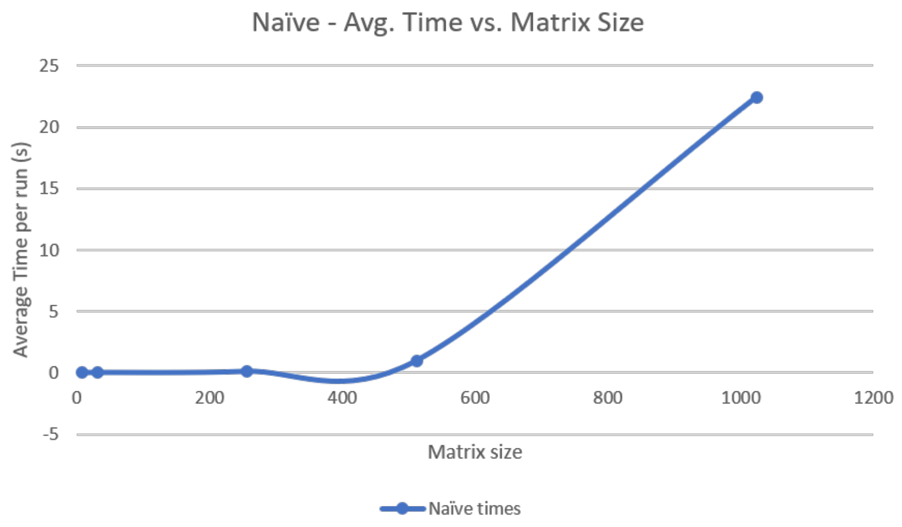
Matrix Size	8	32	256	512	1024
Avg. Time (s)	0.000003	0.000171	0.09249	0.96359	22.4469

Figure 1: Naive - Log of runtime vs. Log of size of matrix



In the naive graphs, we can see a clear exponential relation between the size of the matrix and the runtime of matrix multiplication. Even more interestingly, a log-log graph for the runtime vs. matrix size of the naive implementation (Fig. 1) shows a clear linear fit. The slight increase in slope of the line between the 512 and 1024 sizes indicates the increase in slow fetches because of the sheer size of the 1024 matrices. We can caluclate this slight uptick by using the cache size of the processor I am using. My computer is equipped with a Intel i7-6600u, with a total of about 4 MB of cache. A 512 by 512 matrix has 262144 values,

Figure 2: Naive - Runtime vs. size of matrix



which equates to $262144 \times 4 = 1048576$ bytes for the integer matrices we're using, or about 1.05 MB per matrix. 3 of these matrices would occupy 3 MB of space, which is less than the capcity of the cache.

However, a 1024 by 1024 integer matrix would require $1024 \times 1024 \times 4 = 4194304$ bytes of space. One matrix alone is already around 4 MB; juggling three matrices would slow the processor down by a noticeable margin. This leads to the slight greater than average increase in logarithmic time required to do the matrix multiplication between 512 and 1024.

Tiling Implementation Results

Execution Time Data:

Size\Tiles	Naive	2	4	8	16	32	64
8	0.000003	0.000005	0.000006				
32	0.000171	0.000235	0.000234	0.000235	0.00025		
256	0.09249	0.11845	0.11687	0.11266	0.11281	0.11375	0.12
512	0.96359	1.30407	1.01014	0.98579	0.98859	0.96876	0.93469
1024	22.4469	11.2656	11.0484	11.0188	8.3219	8.2859	8.6077

Looking at the table above, we can see that the extra control logic necessary for tiling means that matrix multiplication runs slightly slower for smaller matrices. However, large gains in efficiency are realized as the matrix increases in size, as we can see in the efficiency gains in the 1024 by 1024 matrix. For the 512 and 1024 matrices, efficiency gains are realized in certain groups - tiling by 2, 4 and 8 lead to important gains, while tiling even more by 16, 32, and 64 has a smaller but still significant effect (Fig. 3). This is due to the effects of L1 and L2 caches, because with increased tiling the smaller matrix tiles are able to fit into the quicker access L2 and then L1 caches.

Figure 3: Tiled - Runtime vs. size of matrix

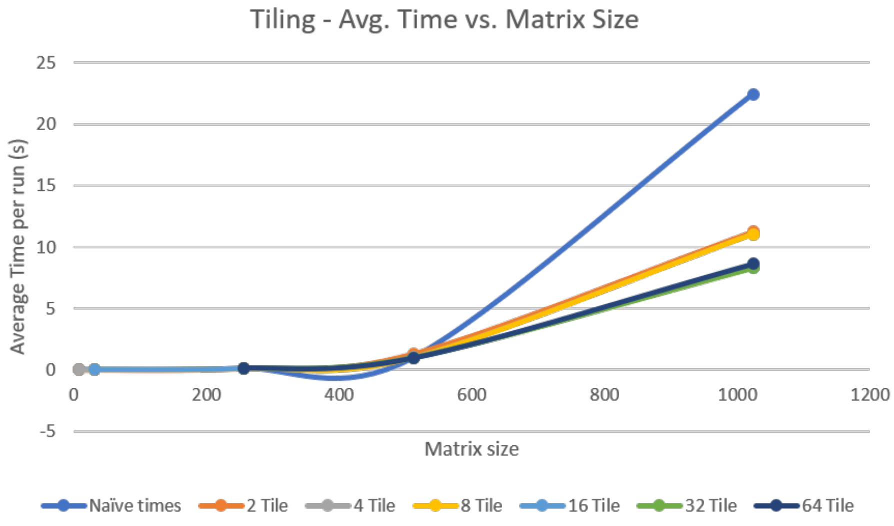
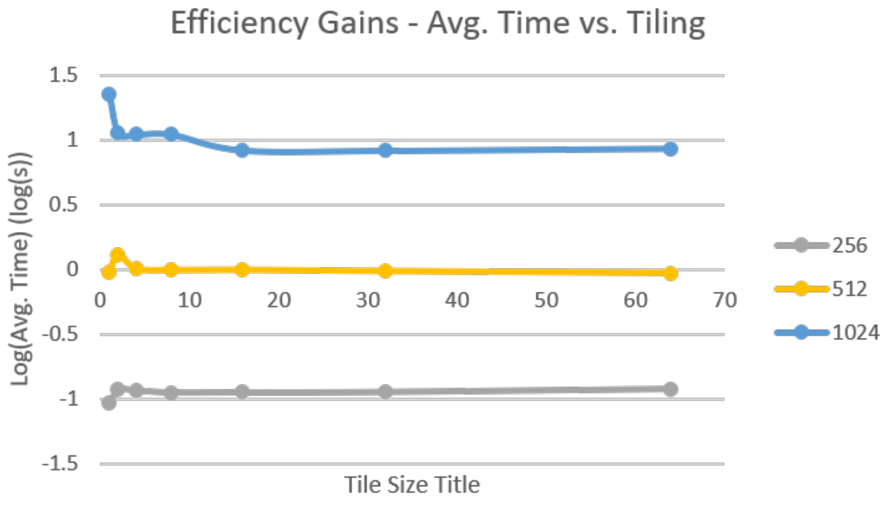


Figure 4: Tiled - Runtime vs. Tiling



It's also interesting to notice how the gains only manifest themselves after a certain threshold is reached, either in terms of numbers of tiles or in the matrix size. In Fig. 4, we can see that the Combination of these two factors lead to realized gains in the 1024 size matrices, while there are smaller improvements in the 512 matrix when you compare the 64 tiling to the naive implementation.

Conclusion

Matrix tiling is a simple and effective method of improving matrix multiplication efficiency. Matrix multiplication is repetitive and requires many slow fetches to the main memory; however, by taking advantage of the repetitive structure and caches that store values closer to the processor we can realize up to 2.5 times improvements in the multiplication of large (more than 1024 by 1024) matrices. While the extra control logic of tiling slows down the multiplication of smaller matrices by a negligible amount, as matrix size and tiling surpasses certain thresholds determine by the cache size of the processor, large efficiency gains can be seen. In machine learning applications, where the multiplication of large matrices forms the fundamental process by which ML training occurs.