Project 1: Cache Simulation Analysis

David Huang

February 7, 2021

1 Correctness

1.1 daxpy

```
python cache-sim.py -d 9 -a daxpy -p
Computation result:
[0.0, 5.0, 10.0, 15.0, 20.0, 25.0, 30.0, 35.0, 40.0]
```

1.2 mxm / mxm_block

```
python cache-sim.py -d 9 -a mxm -p
python cache-sim.py -d 9 -a mxm_block -f 3 -p
Computation result:
[[ 3672.  3744.  3816.  3888.  3960.  4032.  4104.  4176.  4248.]
[ 9504.  9738.  9972.  10206.  10440.  10674.  10908.  11142.  11376.]
[15336.  15732.  16128.  16524.  16920.  17316.  17712.  18108.  18504.]
[21168.  21726.  22284.  22842.  23400.  23958.  24516.  25074.  25632.]
[27000.  27720.  28440.  29160.  29880.  30600.  31320.  32040.  32760.]
[32832.  33714.  34596.  35478.  36360.  37242.  38124.  39006.  39888.]
[38664.  39708.  40752.  41796.  42840.  43884.  44928.  45972.  47016.]
[44496.  45702.  46908.  48114.  49320.  50526.  51732.  52938.  54144.]
[50328.  51696.  53064.  54432.  55800.  57168.  58536.  59904.  61272.]]
```

2 Associativity

My results are reported in Table 1. From the standpoint of the blocked matrix-matrix algorithm, using an 8-way set associative cache appears to be a fine decision. From my results, the overall miss rate improvement peaks and plateaus beginning with a cache associativity of 4, where increased associativity beyond 4 doesn't appear to worsen the read miss and write miss rates for the blocked matrix-matrix algorithm with default settings (besides associativity).

Table 1: Associativity

Cache Associativity	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %	Runtime
1	449,971,200	220,590,543	4,049,457	**1.80%**	3,630,720	516,480	**12.45%**	148.71 secs
2	449,971,200	223,266,764	1,373,236	**0.61%**	4,060,800	86,400	**2.08%**	189.55 secs
4	449,971,200	223,747,200	892,800	**0.40%**	4,060,800	86,400	**2.08%**	175.64 secs
8	449,971,200	223,747,200	892,800	**0.40%**	4,060,800	86,400	**2.08%**	159.98 secs
16	449,971,200	223,747,200	892,800	**0.40%**	4,060,800	86,400	**2.08%**	162.32 secs
1,024	449,971,200	223,747,200	892,800	**0.40%**	4,060,800	86,400	**2.08%**	179.09 secs

3 Memory Block Size

My results are reported in Table 2. The observable trend in my results exhibits a parabolic ”sweet spot” of memory size that yields a minimal miss rate. The best performance (i.e. lowest miss rates) was achieved with a memory block size of 256 bytes. Below, I explain the observed trend.

Table 2: Memory Block Size

Memory Block Size	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %	Runtime
8	449,971,200	216,517,308	8,122,692	**3.62%**	3,456,000	691,200	**16.67%**	178.09 secs
16	449,971,200	220,374,140	4,265,860	**1.90%**	3,801,600	345,600	**8.33%**	153.55 secs
32	449,971,200	222,302,556	2,337,444	**1.04%**	3,974,400	172,800	**4.17%**	153.63 secs
64	449,971,200	223,266,764	1,373,236	**0.61%**	4,060,800	86,400	**2.08%**	151.33 secs
128	449,971,200	223,748,868	891,132	**0.40%**	4,104,000	43,200	**1.04%**	142.96 secs
256	449,971,200	223,989,860	650,140	**0.29%**	4,125,600	21,600	**0.52%**	140.36 secs
512	449,971,200	218,369,320	6,270,680	**2.79%**	3,734,992	412,208	**9.94%**	156.55 secs
1,024	449,971,200	209,335,669	15,304,331	**6.81%**	2,937,064	1,210,136	**29.18%**	183.43 secs

- *Block Size: 8.* Small block sizes provide limited coverage of words, albeit allowing for a larger number of small blocks in the cache. However, due to the limited coverage space of (only fits one word of size 8 bytes), small blocks cannot provide enough locality coverage to remain temporally or spatially "relevant" for very long with respect to the operations of the matrix-matrix algorithm. In other words, each block is only useful for a very tiny fraction of operations within a period of time, requiring cache to retrieve more blocks from RAM as "used" blocks accumulate in in cache. A blocked sub-matrix row requires no fewer than 32 retrievals (assuming data not in cache).

- $\vdots$

- *Block Size: 128.* Increasing the block sizes allows for blocks to contain a larger capacity of words. In this case, the larger capacity of 128 bytes allows the block to strike a nice balance of both temporal and spatial locality of words for the blocked matrix-matrix algorithm. But not the best.

- *Block Size: 256.* The block size of 256 bytes performed the best, exhibiting the overall lowest miss rates in the experiments. Taking a closer look, each block can fit $2^8/2^3 = 32$ words, which matches the blocking factor of the default blocked matrix-matrix algorithm experiment. Assuming perfect alignment, a single retrieval of a single block can provide the entire row of a matrix during the sub-matrix procedure of the blocked MxM algorithm where each sub-matrix has a dimension of 32. Big improvement from block size of 8 bytes, as the 256 bytes memory will be efficiently useful for each of the the sub-matrix computations.

- $\vdots$

- *Block Size: 1024.* Continued increasing of the block sizes eventually leads to a deterioration of performance and efficiency gained from increasing block size capacities from 8 bytes (1 word) to 256 bytes (32 words). When blocks are too large, they may provide more word storage capacity or "locality coverage" than needed for an algorithm at any given point. In this case, the large block sizes end up taking space away from other memory blocks needed for the algorithm's procedures. For the experiments run with a blocking factor of 32, each block contains $2^7 = 128$ words, which cannot provide more utility beyond a quarter of it's stored words for a blocked matrix-matrix multiplication operation with a blocking factor of 32.

4 Total Cache Size

My results are reported in Tables 3 and 3a-3c. With default settings except cache size (32KB), it appears that the 32KB cache size of Skylake is insufficient to meet the 0.5% data read miss rate when running the blocked matrix-matrix algorithm (Table 3). In Table 3a, it appears that the 32KB cache is able to improve its read miss rate by increasing its associativity while keeping other settings default. In Table 3b, it appears that the 32KB cache is able to improve its read miss rate by increasing the memory block size while keeping other settings default. In a grid search of pairwise configurations, Table 3c reveals an optimal architecture for the 32KB cache that performs well beyond the 0.5% data read miss rate threshold (highlighted).

Proposed Architectural Changes w.r.t. the Blocked MxM (blocking factor: 32)

- Associativity: $2 \longrightarrow 8$
- Mem. Block Size: $64 \longrightarrow 256$

Table 3: Total Cache Size

Total Cache Size	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %	Runtime
4,096	449,971,200	96,768,000	127,872,000	**56.92%**	-	4,147,200	**100.00%**	598.34 secs
8,192	449,971,200	195,571,890	29,068,110	**12.94%**	-	4,147,200	**100.00%**	303.00 secs
16,384	449,971,200	216,871,425	7,768,575	**3.46%**	3,225,600	921,600	**22.22%**	196.73 secs
32,768	449,971,200	221,867,190	2,772,810	**1.23%**	4,060,800	86,400	**2.08%**	151.55 secs
65,536	449,971,200	223,266,764	1,373,236	**0.61%**	4,060,800	86,400	**2.08%**	178.75 secs
131,072	449,971,200	223,624,633	1,015,367	**0.45%**	4,060,800	86,400	**2.08%**	168.85 secs
262,144	449,971,200	224,012,790	627,210	**0.28%**	4,060,800	86,400	**2.08%**	147.47 secs
524,288	449,971,200	224,098,454	541,546	**0.24%**	4,060,800	86,400	**2.08%**	153.63 secs

Table 3a: Skylake 32KB Cache Associativity - Default MxM Blocked

Associativity	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %	Runtime
2	449,971,200	221,867,190	2,772,810	**1.23%**	4,060,800	86,400	**2.08%**	160.30 secs
4	449,971,200	223,470,210	1,169,790	**0.52%**	4,060,800	86,400	**2.08%**	164.91 secs
8	449,971,200	223,416,870	1,223,130	**0.54%**	4,060,800	86,400	**2.08%**	163.06 secs
16	449,971,200	223,391,430	1,248,570	**0.56%**	4,060,800	86,400	**2.08%**	158.84 secs
32	449,971,200	223,378,710	1,261,290	**0.56%**	4,060,800	86,400	**2.08%**	172.82 secs

Table 3b: Skylake 32KB Cache Memory Block Size - Default MxM Blocked

Mem. Block	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %	Runtime
64	449,971,200	221,867,190	2,772,810	**1.23%**	4,060,800	86,400	**2.08%**	155.44 secs
128	449,971,200	222,410,130	2,229,870	**0.99%**	4,104,000	43,200	**1.04%**	180.65 secs
256	449,971,200	222,694,200	1,945,800	**0.87%**	4,125,600	21,600	**0.52%**	150.93 secs
512	449,971,200	212,212,111	12,427,889	**5.53%**	3,326,416	820,784	**19.79%**	175.67 secs
1,024	449,971,200	182,741,590	41,898,410	**18.65%**	915,688	3,231,512	**77.92%**	272.84 secs

Table 3c: Skylake 32 KB Cache Memory Configuration Analysis - Default MxM Blocked

Associativity	Mem. Block	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %	Runtime
2	64	449,971,200	221,867,190	2,772,810	**1.23%**	4,060,800	86,400	**2.08%**	148.34 secs
2	128	449,971,200	222,410,130	2,229,870	**0.99%**	4,104,000	43,200	**1.04%**	142.18 secs
2	256	449,971,200	222,694,200	1,945,800	**0.87%**	4,125,600	21,600	**0.52%**	141.08 secs
2	512	449,971,200	212,212,111	12,427,889	**5.53%**	3,326,416	820,784	**19.79%**	176.98 secs
2	1,024	449,971,200	182,741,590	41,898,410	**18.65%**	915,688	3,231,512	**77.92%**	263.99 secs
4	64	449,971,200	223,470,210	1,169,790	**0.52%**	4,060,800	86,400	**2.08%**	146.46 secs
4	128	449,971,200	224,105,610	534,390	**0.24%**	4,104,000	43,200	**1.04%**	145.94 secs
4	256	449,971,200	224,395,380	244,620	**0.11%**	4,125,600	21,600	**0.52%**	142.11 secs
4	512	449,971,200	217,620,000	7,020,000	**3.12%**	4,136,400	10,800	**0.26%**	170.37 secs
4	1,024	449,971,200	169,788,600	54,851,400	**24.42%**	685,800	3,461,400	**83.46%**	319.69 secs
8	64	449,971,200	223,416,870	1,223,130	**0.54%**	4,060,800	86,400	**2.08%**	155.98 secs
8	128	449,971,200	224,052,270	587,730	**0.26%**	4,104,000	43,200	**1.04%**	152.84 secs
8	256	449,971,200	224,411,340	228,660	**0.10%**	4,125,600	21,600	**0.52%**	150.93 secs
8	512	449,971,200	217,620,000	7,020,000	**3.12%**	4,136,400	10,800	**0.26%**	177.01 secs
8	1,024	449,971,200	159,608,400	65,031,600	**28.95%**	685,800	3,461,400	**83.46%**	341.85 secs
16	64	449,971,200	223,391,430	1,248,570	**0.56%**	4,060,800	86,400	**2.08%**	163.77 secs
16	128	449,971,200	224,026,830	613,170	**0.27%**	4,104,000	43,200	**1.04%**	155.54 secs
16	256	449,971,200	224,407,440	232,560	**0.10%**	4,125,600	21,600	**0.52%**	157.46 secs
16	512	449,971,200	217,620,000	7,020,000	**3.12%**	4,136,400	10,800	**0.26%**	179.23 secs
16	1,024	449,971,200	122,947,200	101,692,800	**45.27%**	685,800	3,461,400	**83.46%**	461.58 secs
32	64	449,971,200	223,378,710	1,261,290	**0.56%**	4,060,800	86,400	**2.08%**	158.99 secs
32	128	449,971,200	224,014,110	625,890	**0.28%**	4,104,000	43,200	**1.04%**	157.99 secs
32	256	449,971,200	224,404,320	235,680	**0.10%**	4,125,600	21,600	**0.52%**	156.09 secs
32	512	449,971,200	217,620,000	7,020,000	**3.12%**	4,136,400	10,800	**0.26%**	177.94 secs
32	1,024	449,971,200	107,136,000	117,504,000	**52.31%**	685,800	3,461,400	**83.46%**	504.89 secs

5 Problem Size and Cache Thrashing

5.1

My results are reported in Table 4. Looking at the regular MxM algorithm, there is a large performance gap between problem sizes of 480, 488, and 512. Details below.

Table 4: Associativity = 2

Matrix Dimension	MxM Method	Blocking Factor	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %	Runtime
480	Regular	-	443,520,000	96,768,000	124,646,400	**56.30%**	604,800	316,800	**34.38%**	552.57 secs
480	Blocked	32	449,971,200	223,266,764	1,373,236	**0.61%**	4,060,800	86,400	**2.08%**	146.63 secs
488	Regular	-	466,047,808	217,799,959	14,866,729	**6.39%**	863,272	89,304	**9.38%**	202.64 secs
488	Blocked	8	494,625,088	244,617,625	2,337,703	**0.95%**	15,151,912	89,304	**0.59%**	262.95 secs
512	Regular	-	538,181,632	117,440,512	151,257,088	**56.29%**	-	1,048,576	**100.00%**	714.40 secs
512	Blocked	32	546,045,952	117,440,512	155,189,248	**56.92%**	-	4,980,736	**100.00%**	717.35 secs

Regular matrix-matrix algorithm

- *Problem size: 480.* Performance benchmark. Not great performance. With memory blocks of size 64 bytes containing 8 words, we note that it'll take at least 60 blocks of memory to describe a matrix row, and at 480 blocks of memory to describe a column. Also, we note that with an associativity of 2, that there are 512 mappable set indices for addresses. Lastly, when considering columns, we note that addresses of columns will stride accordingly by multiples of 60 blocks. WLOG, suppose we read in the first column of the matrix. In considering the mapped sets, we can express the mapping with modular arithmetic.

  $$60x \equiv 0 \pmod{512}$$

  Solving for x would effectively mean figuring out the total number of distinct sets all the blocks corresponding to a single column would map to.

  $$60x \equiv 0 \pmod{512}$$
  $$\implies 15 \cdot 2^2 x \equiv 2^9 \pmod{512}$$

  It turns out that the minimum positive solution here is $x = 2^7 = 128$, and in turn means that a single column of 480 blocks would map to only 128 set indices. With an associativity of 2, the 128 sets can only accomodate up to 256 of the 480 blocks before running out of space. Meaning, the cache cannot store a complete matrix column of this problem size. See Figure 1 for a visualized histogram of the above explanation.

- *Problem size: 488.* A huge improvement in performance is observed. Read misses drop by a factor of roughly 8 to 9. In this scenario, one column will take 488 blocks of memory to describe. When considering columns, addresses of columns will stride by multiples of 61 blocks. WLOG, suppose we read in the first column of the matrix. Again, we can express the set mapping with modular arithmetic.

$$61x \equiv 0 \pmod{512}$$

Unlike the previous problem, we can observe that 61 and 512 are coprime, because 61 is prime and 512 is a power of 2. This implies that we can map up to 512 distinct set indices with strides of 61 blocks. For a column size of 488 blocks in the given problem size 488, we get 488 distinct set indicies without any issues of capacity limitation unlike the previous problem size; ergo, the cache can fit an entire matrix column without conflict. See Figure 1 for a visualized histogram of the above explanation.

- *Problem size: 512.* Performance is similar to that of problem size 480. Each column will take 512 blocks of memory to describe. Addresses of column blocks will stride by multiples of in the mapping to 512 set indices.

$$64x \equiv 0 \pmod{512}$$
$$\implies 2^6 x \equiv 2^9 \pmod{512}$$

So we'd expect to see only $x = 2^3 = 8$ distinct set indices for a given column of matrix blocks. Interestingly, this doesn't impact performance as badly as I would've thought. See Figure 2 for a visualized histogram.

In summary, strides w.r.t. both problem size and cache configuration can have a huge effect on the behavior of memory allocation and storage. A good alignment of stride and cache will result in relatively uniform spread of data storage, whereas a poor alignment will result in a "denser" and clustered spread of data across the cache, possibly resulting in the ejection of temporally and spatially relevant data.

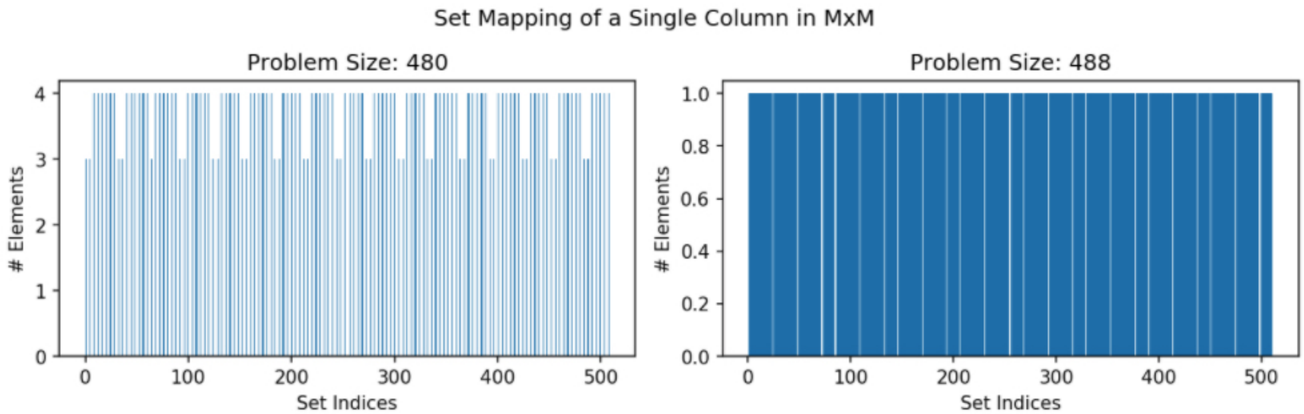


Figure 1: Set mapping histogram of a single column in MxM across two problem sizes: 480 & 488

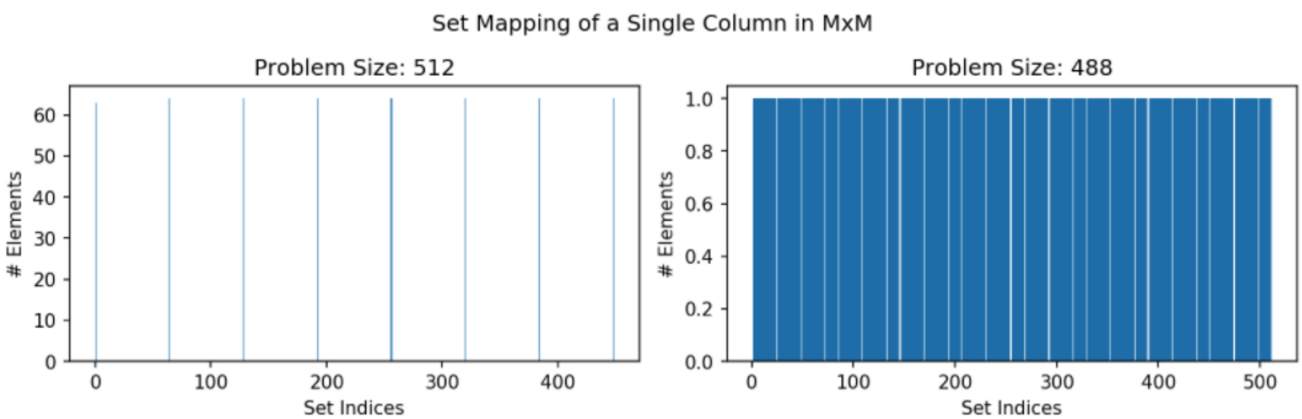
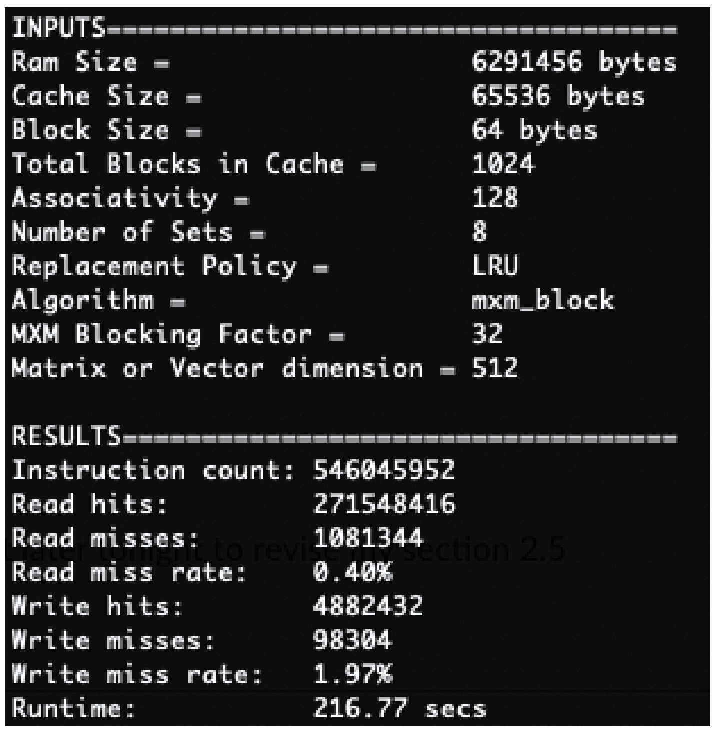


Figure 2: Set mapping histogram of a single column in MxM across two problem sizes: 512 & 488

5.2

The blocked matrix-matrix multiply algorithm is not successful in improving cache performance compared to the regular method for the 512×512 matrix. Following the previous analysis on problem size 512, we see that column-wise blocks of a single column map to only 8 distinct set indices. With a blocking factor of 32, we'd need to store a sub-matrix of 32×32 matrix blocks, or 32×4 memory blocks. With an associativity of 2, we observe that we can only store half of the first sub-matrix (e.g. top half and bottom half, etc.). To store a single submatrix of blocking factor 32×32 matrix blocks, we'd need a cache with associativity of 4. Now to store a whole column of sub-matrices, we can consider the number of 32×32 submatrices in the column, which is $2^9/2^5 = 2^4 = 16$. Thus, we'd need an associativity of $4 \cdot 16 = 64$, to effectively store a whole "column" of 32×32 matrix blocks. Increasing the associativity even more should improve results, allowing for the cache to handle even more set mapping conflicts. From testing, I found that an associativity of 64 to produce good results of 1.31% read miss rate, and an associativity of 128 to produce great results. See Figure 3.


```
INPUTS====================================
Ram Size =                      6291456 bytes
Cache Size =                    65536 bytes
Block Size =                    64 bytes
Total Blocks in Cache =         1024
Associativity =                 128
Number of Sets =                8
Replacement Policy =            LRU
Algorithm =                     mxm_block
MXM Blocking Factor =           32
Matrix or Vector dimension =    512

RESULTS===================================
Instruction count: 546045952
Read hits:         271548416
Read misses:       1081344
Read miss rate:    0.40%
Write hits:        4882432
Write misses:      98304
Write miss rate:   1.97%
Runtime:           216.77 secs
```

Figure 3: Increased associativity of 128 for Blocked MxM of size 512

5.3

My results are reported in Tables 5 and 6. There is a improvement in performance with full associativity. With associativity set to 8, we still observe a poor performance of the blocked MxM algorithm on problem size 512. Despite the desired algorithm performance, a hardware designer may decide against implementing a fully associative cache because every single read requires a complete scan of the cache, which can be quite costly over time. However, I note, my runtimes are not reflective of fully associative speed costs, due to the manner in which I implemented the cache (using hash table lookups).

Table 5: Associativity = 8

Matrix Dimension	MxM Method	Blocking Factor	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %	Runtime
480	Regular	-	443,520,000	96,768,000	124,646,400	**56.30%**	604,800	316,800	**34.38%**	562.91 secs
480	Blocked	32	449,971,200	223,747,200	892,800	**0.40%**	4,060,800	86,400	**2.08%**	162.50 secs
488	Regular	-	466,047,808	217,871,995	14,794,693	**6.36%**	863,272	89,304	**9.38%**	208.53 secs
488	Blocked	8	494,625,088	244,754,972	2,200,356	**0.89%**	15,151,912	89,304	**0.59%**	215.77 secs
512	Regular	-	538,181,632	117,440,512	151,257,088	**56.29%**	688,128	360,448	**34.38%**	649.44 secs
512	Blocked	32	546,045,952	117,440,512	155,189,248	**56.92%**	688,128	4,292,608	**86.18%**	763.11 secs

Table 6: Full Associativity = 1024

Matrix Dimension	MxM Method	Blocking Factor	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %	Runtime
480	Regular	-	443,520,000	207,331,202	14,083,198	**6.36%**	835,200	86,400	**9.38%**	242.32 secs
480	Blocked	32	449,971,200	223,747,200	892,800	**0.40%**	4,060,800	86,400	**2.08%**	199.58 secs
488	Regular	-	466,047,808	217,871,994	14,794,694	**6.36%**	863,272	89,304	**9.38%**	252.27 secs
488	Blocked	8	494,625,088	245,079,944	1,875,384	**0.76%**	15,151,912	89,304	**0.59%**	249.43 secs
512	Regular	-	538,181,632	251,625,474	17,072,126	**6.35%**	950,272	98,304	**9.38%**	278.64 secs
512	Blocked	32	546,045,952	271,548,416	1,081,344	**0.40%**	4,882,432	98,304	**1.97%**	235.78 secs

5.4

Two possible software optimizations to improve the cache performance of blocked MxM:

- One can store the transpose of one of the two matrices. Because data is stored as lines of lines, the column elements that are normally dispersed across D blocks (for problem size D) will better fit within the memory blocks, requiring fewer blocks to sit in memory. In other words, when considering columns, the transposed matrix will result in far more efficient utilization of memory blocks. Along with this modification, the algorithm would need to be modified to operate on two rows of matrices (one being transposed).

- One can also try reducing the blocking factor down from 32 to 8 to reduce the cache conflicts that arise when storing memory blocks of the sub-matrices. Upon testing, I observed improvements in miss rates. For a blocking factor 16, a roughly read 15% miss rate. And for a blocking factor of 8, a roughly 1.5% read miss rate. See Figure 3.

```
INPUTS====================================
Ram Size =                     6291456 bytes
Cache Size =                   65536 bytes
Block Size =                   64 bytes
Total Blocks in Cache =        1024
Associativity =                2
Number of Sets =               512
Replacement Policy =           LRU
Algorithm =                    mxm_block
MXM Blocking Factor =          8
Matrix or Vector dimension =   512

RESULTS===================================
Instruction count: 571211776
Read hits:         280720384
Read misses:       4492288
Read miss rate:    1.58%
Write hits:        16515072
Write misses:      1048576
Write miss rate:   5.97%
Runtime:           234.81 secs
```

Figure 4: Reduced blocking factor of 16 for Blocked MxM of size 512

6 Replacement Policy

My results are reported in Tables 7a-7c. I tested each of the three replacement strategies across all three algorithms: daxpy, MxM, and Blocked MxM. I detail replacement strategy performance by algorithm below.

- *daxpy.* No strategy here outperforms as best. This makes sense, because elements of the vectors are not re-used more than once. The manner in which blocks are replaced won't affect cache performance. I suspect that performance can only be improved here not by replacement strategy, but instead by some predictive block retrieval strategy.

Table 7a: daxpy								
Replacement Strategy	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %	Runtime
random	80,000,000	17,500,000	2,500,000	**12.50%**	35,000,000	5,000,000	**12.50%**	153.88 secs
FIFO	80,000,000	17,500,000	2,500,000	**12.50%**	35,000,000	5,000,000	**12.50%**	140.22 secs
LRU	80,000,000	17,500,000	2,500,000	**12.50%**	35,000,000	5,000,000	**12.50%**	157.98 secs

Table 7b: MxM								
Replacement Strategy	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %	Runtime
random	443,520,000	115,048,742	106,365,658	**48.04%**	625,174	296,426	**32.16%**	617.81 secs
FIFO	443,520,000	96,768,000	124,646,400	**56.30%**	604,800	316,800	**34.38%**	491.96 secs
LRU	443,520,000	96,768,000	124,646,400	**56.30%**	604,800	316,800	**34.38%**	502.51 secs

Table 7c: MxM Blocked								
Replacement Strategy	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %	Runtime
random	449,971,200	222,906,439	1,733,561	**0.77%**	4,005,116	142,084	**3.43%**	116.26 secs
FIFO	449,971,200	223,046,909	1,593,091	**0.71%**	4,059,466	87,734	**2.12%**	110.64 secs
LRU	449,971,200	223,266,764	1,373,236	**0.61%**	4,060,800	86,400	**2.08%**	159.53 secs

- *MxM.* Interestingly, the random replacement strategy performed best here. There's no difference between LRU and FIFO, because they effectively operate with the same behavior as elements are accessed sequentially, both strategies will eject the earliest-accessed block. I suspect random performs best because the random ejection of a memory block can sometimes yield a "good" ejection where we won't eject the earliest-accessed block. Keeping the earliest-accessed block line, suppose the first row of a column for the vector-matrix subprocess, allows the block line to be re-used for multiple column iterations, thereby reducing the miss rate.

- *Blocked MxM.* Across a few runs, it appears that LRU appears to perform best, FIFO second, and random the worst. This makes intuitive sense as LRU optimizes upon temporal locality. When the matrix operation is blocked, the cache will tend to keep the most relevant data demanded by the sub-matrix multiplication within cache, whereas random may eject useful blocks, and FIFO may also ejected useful blocks. The blocked matrix algorithm will re-use a single sub-matrix block of 32×32 many times, and FIFO won't respect always respect the algorithm's temporal focus on a blocked sub-matrix.

In summary, I think that with a smart algorithm that optimizes both spatial and temporal locality, one cannot go wrong pairing it with a temporally focused replacement strategy like LRU. However, I can see how a random replacement strategies can be useful when when memory accesses are totally random or fit some predictable probability distribution, like a stochastic simulation of sorts.