AA: Project-1

Isaac Hirsch

April 2025

1 Introduction

Isaac Hirsch
ihirsch@uchicago.edu

2 Usage

The standard script (i.e. everything but the l2 stuff) can be run by the command 'python3 cache-sim.py standard flags' with Python 3.8.10 or Python 3.12.2. For faster runs do 'pip install numba'.
The l2 program is in the two-level-cache directory and can also be run by 'python3 cache-sim.py standard flags' where the flags effects the l1 cache.

3 Correctness

Correctness tests can be ran by adding the -t, -p, and -a algorithm flags to the end of the cache-sim.py script.

$$\mathbf{mxm\_blocked} = \begin{bmatrix} 30 & 36 & 42 \\ 84 & 108 & 132 \\ 138 & 180 & 222 \end{bmatrix}$$

$$\mathbf{mxm} = \begin{bmatrix} 30 & 36 & 42 \\ 84 & 108 & 132 \\ 138 & 180 & 222 \end{bmatrix}$$

$$\mathbf{daxpy} = \begin{bmatrix} 0 & 5 & 10 & 15 & 20 & 25 & 30 & 35 & 40 \end{bmatrix}$$

4 Associativity

My results in Table 1 suggest that this is a reasonable result, as long as 8-way associativity does not increase CPI or complexity significantly over 2-way, since both had almost identical results.

Cache Associativity	**Instructions**	**Read Hits**	**Read Misses**	**Read Miss %**	**Write Hits**	**Write Misses**	**Write Miss %**
1	449971200	222216486	2423514	1.08	3630720	516480	12.45
2	449971200	223698840	941160	0.42	4060800	86400	2.08
4	449971200	223747200	892800	0.40	4060800	86400	2.08
8	449971200	223747200	892800	0.40	4060800	86400	2.08
16	449971200	223747200	892800	0.40	4060800	86400	2.08
1024 (Fully Associative)	449971200	223747200	892800	0.40	4060800	86400	2.08

Table 1: Associativity table

5 Memory Block Size

In table 2 he read and write miss rates drop until 256 and then start increasing again. This is exactly as expected since with a 32x32 tiling scheme of 8-byte doubles, each tile row fits exactly into a 256 byte block.
If you use a smaller block size, you will have multiple misses per row which is what we see here with the read miss rate having every time we double block size up until 256 since we need double the number of reads from memory per row.
If you use a larger block size, you will gain no benefits in terms of pre-loading elements, and you will loose out on the number of sets which will cause increased conflict and capacity misses.

Block Size (bytes)	**Instructions**	**Read Hits**	**Read Misses**	**Read Miss %**	**Write Hits**	**Write Misses**	**Write Miss %**
8	449971200	217110720	7529280	3.35	3456000	691200	16.67
16	449971200	220875360	3764640	1.68	3801600	345600	8.33
32	449971200	222757680	1882320	0.84	3974400	172800	4.17
64	449971200	223698840	941160	0.42	4060800	86400	2.08
128	449971200	224169420	470580	0.21	4104000	43200	1.04
256	449971200	224404710	235290	0.10	4125600	21600	0.52
512	449971200	222728738	1911262	0.85	3734992	412208	9.94
1024	449971200	219352185	5287815	2.35	2932456	1214744	29.29

Table 2: Memory Block size table

6 Total Cache Size

I found that 32KB was just barely above a read miss rate of 0.5%, while 64 KB was bellow it. To achieve a ¡0.5% miss rate with 32KB, I simply doubled the block size (as per my last section) and achieved a read miss rate of 0.32

7 Problem Size and Cache Thrashing

1. Cache performance does vary a lot throughout Table 4. This is due to conflict misses in B, where either the fact that n*n in the case of naive mxm or n in the case of tiled mxm is a multiple of the number of sets and therefore causes each element of B to be loaded n times.

Cache Size (bytes)	**Instructions**	**Read Hits**	**Read Misses**	**Read Miss %**	**Write Hits**	**Write Misses**	**Write Miss %**
4096	449971200	109688400	114951600	51.17	0	4147200	100.00
8192	449971200	206585790	18054210	8.04	0	4147200	100.00
16384	449971200	222907245	1732755	0.77	3225600	921600	22.22
32768	449971200	223321204	1318796	0.59	3946112	201088	4.85
32768 (128 byte blocks)	449971200	223913696	726304	0.32	3989312	157888	3.81
65536	449971200	223698840	941160	0.42	4060800	86400	2.08
131072	449971200	223736248	903752	0.40	4060800	86400	2.08
262144	449971200	224019377	620623	0.28	4060800	86400	2.08
524288	449971200	224133224	506776	0.23	4060800	86400	2.08

Table 3: Total Cache size table

2. 512 destroys the performance of tiled matrix multiplication due to conflict misses. Specifically, since it is the only one the number of sets (1024) is a multiple of the dimension, every other element in a column maps to the same set which causes conflict misses to erase all of the gains we got from using tiled matrix multiplication, and causes every element of B to be loaded n times (same as naive).

3. I noticed only small improvements in miss rates for the 8-way associative version in Table 5 which makes sense since mxm and mxm tiled cause O(-d) and O(-f) conflicts in B, respectively. These are both way bigger than 8 conflicts and therefore we see basically no gain. Fully associative in Table 6 on the other hand fixes the problem entirely since it increases the associativity past the number of elements trying to map to each set in each calculation of a C[i,j]. However, in real life fully-associative data caches are often not popular due to the complexity of implementation and the penalty paid in CPI.

4. Option 1: Pad the matracies with an extra row and column of 0s, and then either not calculate or remove this row and column from C (the result) after the calculation. This slight alteration makes it so that n/nxn is not a divider/multiple of the number of sets, and therefore you will not run into significant conflict misses. Option 2 is to only store the rows in continuous memory, but have the matrix be an array of n pointers to those rows (which are scattered in memory). Both of these should fix the conflict issue.

Matrix Dimension	**MxM Method**	**Blocking Factor**	**Instructions**	**Read Hits**	**Read Misses**	**Read Miss %**	**Write Hits**	**Write Misses**	**Write Miss %**
480	Regular	-	443520000	107236881	114177519	51.57	604800	316800	34.38
480	Blocked	32	449971200	223698840	941160	0.42	4060800	86400	2.08
488	Regular	-	466047808	217673225	14993463	6.44	863272	89304	9.38
488	Blocked	8	494625088	244625824	2329504	0.94	15151912	89304	0.59
512	Regular	-	538181632	134120448	134577152	50.08	0	1048576	100.00
512	Blocked	32	546045952	137084928	135544832	49.72	0	4980736	100.00

Table 4: Matrix Multiply Problem size table – Associativity = 2

Matrix Dimension	**MxM Method**	**Blocking Factor**	**Instructions**	**Read Hits**	**Read Misses**	**Read Miss %**	**Write Hits**	**Write Misses**	**Write Miss %**
480	Regular	-	443520000	107317995	114096405	51.53	604800	316800	34.38
480	Blocked	32	449971200	223747200	892800	0.40	4060800	86400	2.08
488	Regular	-	466047808	218080372	14586316	6.27	863272	89304	9.38
488	Blocked	8	494625088	244751209	2204119	0.89	15151912	89304	0.59
512	Regular	-	538181632	134121472	134576128	50.08	688128	360448	34.38
512	Blocked	32	546045952	137101312	135528448	49.71	688128	4292608	86.18

Table 5: Matrix Multiply Problem size table – Associativity = 8

Matrix Dimension	**MxM Method**	**Blocking Factor**	**Instructions**	**Read Hits**	**Read Misses**	**Read Miss %**	**Write Hits**	**Write Misses**	**Write Miss %**
480	Regular	-	443520000	207532802	13881598	6.27	835200	86400	9.38
480	Blocked	32	449971200	223747200	892800	0.40	4060800	86400	2.08
488	Regular	-	466047808	218080370	14586318	6.27	863272	89304	9.38
488	Blocked	8	494625088	245079944	1875384	0.76	15151912	89304	0.59
512	Regular	-	538181632	251854850	16842750	6.27	950272	98304	9.38
512	Blocked	32	546045952	271548416	1081344	0.40	4882432	98304	1.97

Table 6: Matrix Multiply Problem size table – Fully Associative

8 Replacement Policy

In Table 7 we can see that LRU has the best performance in terms of both read and write hit rates. FIFO is behind it, and Random is in last place.

Replacement Policy	**Instructions**	**Read Hits**	**Read Misses**	**Read Miss %**	**Write Hits**	**Write Misses**	**Write Miss %**
Random	449971200	223494091	1145909	0.51	4037137	110063	2.65
FIFO	449971200	223629200	1010800	0.45	4059340	87860	2.12
LRU	449971200	223698840	941160	0.42	4060800	86400	2.08

Table 7: Replacement policy table

9 Second-Level Cache Extension

Config	**L1 Read Miss %**	**L1 Write Miss %**	**L2 Read Miss %**	**L2 Write Miss %**
L1 Only	0.20	0.04	N/A	N/A
L1 + L2	0.20	0.04	0.05	0.00

Table 8: Miss rates at each cache level

Config	**RAM Accesses**	**Read Hits (L1+L2)**	**Write Hits (L1+L2)**
L1 Only	1002300	223724100	4060800
L1 + L2	122400	224604000	4579200

Table 9: Memory access totals and hits with and without L2

10 Limitations

The only limitation that I am aware of is that my L2 data cache's statistics are somewhat confusing. Specifically since during l1 write misses, l1 first loads the data before writing it to cache, l2 will always load the data before l1 writes to l2 so it is impossible for l2 to have a write miss. This also means that the loads done by l2 aren't just during l1 load misses, but also l1 write misses. Because of this it is somewhat hard to interpret the statistics from l2.