2.1 Correctness

9 daxpy

0	5	10	15	20	25	30	35	40

9x9 matrix-matrix multiply

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

2.2 Associativity

Purely from the standpoint of our blocked matrix-matrix multiply algorithm, 8-way set associativity is not the optimal design decision. As seen in the table below, I received the same results for associativities 2 and up. Therefore, the only difference is in speed the algorithm takes. A 2-way set associative cache is a lot faster than those with higher set associativities (all other things being equal) as only a maximum of 2 checks need to be made when searching for a block within a given set.

Table 1:

Cache Associativity	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
1	449971200	222347265	2292735	1.0	3630720	516480	12.5
2	449971200	223747200	892800	0.4	4060800	86400	2.1
4	449971200	223747200	892800	0.4	4060800	86400	2.1
8	449971200	223747200	892800	0.4	4060800	86400	2.1
16	449971200	223747200	892800	0.4	4060800	86400	2.1
1024	449971200	223747200	892800	0.4	4060800	86400	2.1

2.3 Memory Block Size

Larger block size reduce the miss rate because of increased spatial locality. However, this has diminishing returns as with larger blocks, fewer blocks overall can be stored in the cache. This in turn results in reduced temporal locality (and a greater miss penalty). As can been seen in table 2 below, the

optimal block size for our default configuration appears to be 256 bytes, the size which benefits greatest from both spatial and temporal locality.

Table 2:

Block Size (bytes)	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
8	449971200	217497600	7142400	3.2	3456000	691200	16.7
16	449971200	221068800	3571200	1.6	3801600	345600	8.3
32	449971200	222854400	1785600	0.8	3974400	172800	4.2
64	449971200	223747200	892800	0.4	4060800	86400	2.1
128	449971200	224193600	446400	0.2	4104000	43200	1.0
256	449971200	224416800	223200	0.1	4125600	21600	0.5
512	449971200	222758283	1881717	0.8	3734992	412208	9.9
1024	449971200	219430764	5209236	2.3	2932456	1214744	29.3

2.4 Total Cache Size

According to my emulator using the default settings, a cache size of 32768 is just below the 0.5% data read miss rate threshold (0.4946%). As we can see in the table, further reducing the cache size to 16384 results a read miss rate greater than 0.5% threshold (0.5927%). Although it is not required for my emulator, increasing the associativity can further reduce the read miss rate if the configuration of the cache and the algorithm can benefit from such a setup. As an example, increasing the associativity from 2 to 4 in my simulator of size 32768 reduced the read miss rate from 0.4946% to 0.397%.

Table 3:

Cache Size (bytes)	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
4096	449971200	109688400	114951600	51.2	0	4147200	100
8192	449971200	206894686	17745314	7.9	0	4147200	100
16384	449971200	223308555	1331445	0.6	3225600	921600	22.2
32768	449971200	223528849	1111151	0.5	3946112	201088	4.8
65536	449971200	223747200	892800	0.4	4060800	86400	2.1
131072	449971200	223749352	890648	0.4	4060800	86400	2.1
262144	449971200	224034196	605804	0.3	4060800	86400	2.1
524288	449971200	224151184	488816	0.2	4060800	86400	2.1

2.5 Problem Size and Cache Thrashing

1) No, there are quite large differences between the three-regular matrix-matrix multiplication results in Table 4. Because of the way the index of an address is calculated, 488 can take advantage of temporal

locality where 480 & 512 cannot. We can see this by noticing that the read miss rate for 488 is 6.4%, almost exactly 1/8th of the 480 and 512 results. If nothing else, this suggests that one block is evicted from that cache every 8 iterations in the 488-case compared to every iteration in the other cases. In each iteration of the matrix-matrix multiplication algorithm's inner loop, the second loaded value's index is increased by the dimension divided by 8. The values are 60, 61 & 64 for 480, 488 & 512 respectively. This value modulus 128 (the number of sets) then gives us the actual cache index. Given the number of sets is 128, incrementing by a number like 60 or 64 practically ensures a poor utilization of the cache due to overlapping cache indexes. By contrast, incrementing by 61 (which happens to be a prime number) minimizes overlapping indices, resulting in a more efficient utilization of the space within the cache.

2) No, blocking makes little difference to the 512 x 512 table in Table 4. The issue with 512x512 is that the number of sets in our default configuration is equal to the dimension of the table (512). Because of this, unlike 480x480 & 488x488, every address within a given iteration of the inner loop maps to the same 2 sets. In the blocked matrix-matrix multiplication case these set indices are 0 & 64 in the first iteration, 1 & 65 in the second, 2 & 66 in the third etc. In the regular matrix-matrix multiplication case, the second index simply alternates between 0 & 64, then 1 & 65 etc. In both cases the second address contains a tag different to the current tag of the block in the 2nd half of the cache set. Because of the eviction policy, on every iteration, the first address is read (a hit) and the second address is a miss leading to the set's 2nd block being evicted. This happens on every iteration, first a hit then a miss, which is why the read miss rate is almost exactly 50%. This logic also explains the 100% write miss rate as when a block which contains the solution address is loaded into the cache as by the time the solution for that address is computed, its block has been evicted from the cache.

3) There is a noticeable improvement between the worst performing configurations in table 5 and those same configurations' results in table 6. This suggests that cache space is being poorly utilized when too tightly mapped to memory locations in those examples with very bad performance in table 5. As established in part 2 of this question, the same cache blocks are replaced with a high frequency (1 out of every 2 reads) in the 512x512 case. Fully associative caches, however (assuming we use a suitable replacement policy) enable us to evict blocks that were not just recently loaded into the cache and are most likely far away from where most of our read and writes are currently taking place. This then allows us to avoid a conflict miss in the next iteration of the inner loop, which explains the significant reduction in miss rates in the fully associative examples. Fully associative caches allow the cache to take better advantage of temporal locality.

Fully associative caches are very slow in our simulator. On every read/write, every block within the cache must be searched until a matching tag is found. In the worst case, on a miss, 1024 comparisons must be made. By comparison, set associative caches share many of the temporal locality benefits that come with a fully associative cache while being almost as fast as a direct mapped cache. In fact, because of the temporal locality benefits, set associative caches in the real world are in the general case faster than direct mapped caches. This is because the penalty incurred on a miss is much greater than the slowdown incurred from searching within a small cache set.

4) Use a blocking matrix multiplication algorithm with a small blocking factor (8 seems to be a good number). Increase the cache size and the block size (524288 with block size 128 produces good results).

Table 4: Associativity = 2

Matrix Dimension	MxM Method	Blocking Factor	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss%
480	Regular	-	443520000	107236881	114177519	51.6	604800	316800	34.4
480	Blocked	32	449971200	223747200	892800	0.4	4060800	86400	2.1
488	Regular	-	466047808	217673225	14993463	6.4	863272	89304	9.4
488	Blocked	8	494625088	244636420	2318908	0.9	15151912	89304	0.6
512	Regular	-	538181632	134120448	134577152	50.1	0	1048576	100
512	Blocked	32	546045952	137084928	135544832	49.7	0	490736	100

Table 5: Associativity = 8

Matrix Dimension	MxM Method	Blocking Factor	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss%
480	Regular	-	443520000	107317995	114096405	51.5	604800	316800	34.4
480	Blocked	32	449971200	223747200	892800	0.4	4060800	86400	2.1
488	Regular	-	466047808	218080372	14586316	6.3	863272	89304	9.4
488	Blocked	8	494625088	244777534	2177794	0.9	15151912	89304	0.6
512	Regular	-	538181632	134121472	134576128	50.1	688128	360448	34.4
512	Blocked	32	546045952	137101312	135528448	49.7	688128	4292608	86.2

Table 6: Fully Associative

Matrix Dimension	MxM Method	Blocking Factor	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss%
480	Regular	-	443520000	207532802	13881598	6.3	835200	86400	9.4
480	Blocked	32	449971200	223747200	89280	0.4	4060800	86400	2.1
488	Regular	-	466047808	218080370	14586318	6.3	863272	89304	9.4
488	Blocked	8	494625088	245079944	1875384	0.8	15151912	89304	0.6
512	Regular	-	538181632	251854850	16842750	6.3	950272	98304	9.4
512	Blocked	32	546045952	271548416	1081344	0.4	4882432	98304	2.0

2.6 Replacement Policy

Best performance: LRU

Table 7:

Replacement Policy	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
Random	449971200	223570732	1069268	0.5	4024632	122568	3.0
FIFO	449971200	223645260	994740	0.4	4060800	86400	2.1
LRU	449971200	223747200	892800	0.4	4060800	86400	2.1