Cache Based Matrix Multiplication

Naman Goyal (2015CSB1021)

Indian Institute of Technology, Ropar

Supervised by

Dr. Neeraj Goel

Table of Contents

1 Introduction

The memory hierarchy of the system follows two level of access. First level is a fast access called the L1 cache. It occurs on a cache hit. Other level is a slow access to main memory which occurs on a cache miss. The cache model can't work optimally without some help from the programmer. The aim is to optimize the matrix multiplication for this model and test the same for different configurations of L1 cache. Cache hit/miss statistics were analyzed using DineroIV (a cache simulator) for various configurations of data cache on two sizes of matrix 200x200 and 500x500. Submission 1 analyzes effect of various cache configurations on miss rate. Submission 2 analyzes effect of cache based matrix multiplication optimization.

2 Submission 1

2.1 Aim

To analyze effect of cache configuration on miss rate using the following configurations

- Cache size: 16kb, 32kb
- Line/Block size: 32 bytes, 64 bytes
- Associativity: 1, 2, 4
- Replacement policy: LRU

2.2 Theory

Misses can be classified as compulsory, capacity and conflict misses. Compulsory misses are cold misses which happen when data is brought into cache for first time. Capacity misses occur when amount of memory required by program is more than cache size. Conflict misses arise due to direct mapped and set associative cache if the working set of program contains blocks which map to same set, but the number of possible lines in a set are less.

Larger block size can help reduce *compulsory misses* since more data can be bought into the cache simultaneously from lower level. However, increasing the block size reduces number of blocks

that can be saved and also its takes more time to read and transfer big blocks from lower level of memory system.

Increasing the cache size, decreases the *capacity misses* since a larger working set of data of can be store at a given time in the cache. But, increasing size of cache requires more area, slows the cache and increases power consumption.

Increasing the associativity of cache reduces *conflict misses* since more blocks can now map to same set. But, it also increases latency and power consumption.

2.3 Statistics

The following table records the cache statistics for 200x 200 matrix.

Demand Miss rate

Cache Size	16 kb			32kb		
Associativity	1	2	4	1	2	4
32 bytes Block	0.1081	0.0655	0.063	0.0855	0.0634	0.063
64 bytes Block	0.4311	0.1076	0.0732	0.4206	0.0767	0.0315

Compulsory misses

Cache Size	16 kb			32kb		
Associativity	1	2	4	1	2	4
32 bytes Block	15000	15000	15000	15000	15000	15000
64 bytes Block	7500	7500	7500	7500	7500	7500

Capacity misses

Cache Size	16 kb			32kb		
Associativity	1	2	4	1	2	4
32 bytes Block	995000	995000	995000	995000	995000	995000
64 bytes Block	497600	497600	497600	497600	497600	497600

Conflict misses

Cache Size	16 kb			32kb		
Associativity	1	2	4	1	2	4
32 bytes Block	723762	40767	131	361840	6278	0
64 bytes Block	6410041	1220075	668562	6242004	725830	844

The following table records the cache statistics for 500x 500 matrix.

Demand Miss rate

Cache Size	16 kb			32kb		
Associativity	1	2	4	1	2	4
32 bytes Block	0.4032	0.2333	0.2958	0.3721	0.1234	0.0627
64 bytes Block	0.5019	0.5313	0.5325	0.4438	0.2124	0.177

Compulsory misses

Cache Size	16 kb			32kb		
Associativity	1	2	4	1	2	4
32 bytes Block	93751	93751	93751	93751	93751	93751
64 bytes Block	46876	46876	46876	46876	46876	46876

Capacity misses

Cache Size	16 kb			32kb		
Associativity	1	2	4	1	2	4
32 bytes Block	100598255	58281211	73922411	15594499	15594499	15594499
64 bytes Block	125090954	132920861	133202739	110796710	53108704	44253251

Conflict misses

Cache Size	16 kb			32kb		
Associativity	1	2	4	1	2	4
32 bytes Block	213396	0	0	77425245	15190876	6512
64 bytes Block	457046	0	0	228468	0	0

2.4 Observations & Discussions

Following points were observed

Observation 1: **Increasing Block size decreases compulsory misses**

Explanation: The observation is inline with theory for the both matrix sizes. There almost 50% reduction in compulsory misses by increasing block size.

Observation 2: **Increasing Block size decreases capacity misses for smaller size matrix but increases capacity misses for larger size matrix**

Explanation: The effect may be explained by the fact that for smaller matrix capacity misses reduce as working set of program is smaller so increases block size increasing preference for spatial locality and thus decreases capacity misses. But for larger program increasing block size reduces the number of blocks that can be saved hence increase capacity misses. Hence increasing block size beyond a certain limit can have negative effects too.

Observation 3: **Increasing cache size decreases capacity misses**

Explanation: The effect is inline with theory but it is seen only for larger size matrix. The possible reason increasing cache size won't provide much gain beyond a certain point. This optimal point should be lesser for smaller program. Hence, for 200 size matrix the gain is nil for increasing the cache size as this optimal point would have been reached for 16KB cache size itself.

Observation 4: **Increasing associativity decreases conflict misses**

Explanation: The effect is inline with theory and seen for the both matrix sizes.

Observation 5: **Hit rate is directly related to cache size and associativity**

Explanation: It may directly be inferred form observation 3 & 4 as reducing miss increases hit rate

Observation 6: **Hit rate is directly related to block size for smaller matrix but inversely for larger size matrix**

Explanation: Due to observation 1, block size effect on hit rate is positive for both matrix sizes. However, due to observation 2, block size has effect on hit rate is possible for smaller matrix and negative for larger matrix. The combined effect is observed i.e. positive for smaller and negative for larger. The observation 2 weighs over observation 1 since number of capacity misses in general are much larger than compulsory misses.

Observation 7: **The best possible configuration for 200 size matrix is 32KB cache, 64 bytes Block Size and associativity 4 while for 500 size matrix is 32KB cache, 32 bytes Block Size and associativity 4**

Explanation: This can be directly inferred from combining observations 5 & 6.

3 Submission 2

3.1 Aim

To analyze effect of cache based matrix multiplication optimization

3.2 Theory

The cache exploits principle of locality which two forms:

[1] Temporal Locality: says that if any memory location is accessed now, it is more likely to be accessed again in near future.

[2] Spatial Locality: says that if any memory location is accessed now, then it's neighboring memory locations are expected to be accessed in near future.

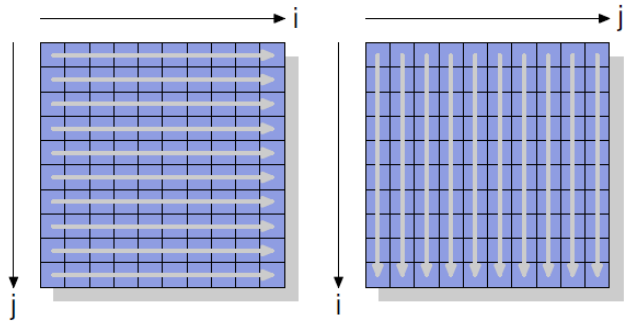
We assume matrices are stored row major. Hence accessing row wise leads to more cache hits due to spatial locality while column wise access lead to more misses as spatial locality is unexploited and we have to load new cell from RAM for all new access.

The straight forward approach of matrix multiplication is

Implementation (mm.c)

```
for(i=0; i<SIZE; i++)
        for(j=0; j<SIZE; j++)
                sum = 0;
                for (k=0; k< SIZE; k++)
                        sum += A[i][k]*B[k][j];
                C[i][j] = sum;
```

The fallacy with this approach is that accesses to B are made column wise hence spatial locality due to storing matrices as row major in C is unexploited.



Matrix Access Pattern

There are three approach to optimize this approach

I. Transpose
II. Loop Reordering
III. Blocking

Transpose

We make another matrix T which transpose of B and access T row-wise while multiplying

This requires touching additional memory, but this cost is, recovered since the accesses per column are more expensive.

Implementation (tranpose.c)

```
//Transposing B Matrix into extra memory
for (i=0; i< SIZE; i++)
        for (j=0; j < SIZE; j++)
                T[j][i] = B[i][j];


for(i=0; i<SIZE; i++)
        for(j=0; j<SIZE; j++)
                sum = 0;
                for (k=0; k< SIZE; k++)
                        sum += A[i][k]*T[j][k];
                C[i][j] = sum;
```

Loop Reordering

We order the loops i – j – k to i – k – j. The result remains computationally correct while accessing matrix B becomes row-wise. Also, no extra memory is required as in case of transpose.

Implementation (reordering.c)

```
// Reordering loop to i - k -j
for(i=0; i<SIZE; i++)
        for(k=0; k<SIZE; k++)
                for (j=0; j< SIZE; j++)
                        C[i][j] += A[i][k]*B[k][j];
```

Blocking

The idea is to break matrix multiplication of larger matrix into smaller sub matrix which is cache specific. The size of sub matrix set such that temporal locality is exploited to maximum. The size is taken as Block Size/ Size of integer.

Here the bigger matrix is broken into small blocks and while accessing these blocks it is guaranteed that once data is loaded it is reused before evicting it. The idea comes from the order in which the elements of matrix B are accessed is: (0,0), (1,0), . .. , (N-1,0), (0,1), (1,1), . . . in the original approach. The elements (0,0) and (0,1) are in the same cache line but, by the time the inner loop completes one round, this cache line has long been evicted. Hence, we handle multiple iterations simultaneously and extend it to matrix C.

Implementation (blocking.c)

```
int BS = 32/ sizeof(int);
for(I=0; I<SIZE; I+=BS )
        for (J=0; J< SIZE; J+=BS)
                for(K=0; K<SIZE; K+= BS)
                        for(i=I; i<min(I+BS, SIZE); i++)
                                for(j=J; j<min(J+BS, SIZE); j++)
                                        for (k=K; k< min(K+BS, SIZE); k++)
                                                C[i][j] += A[i][k]*B[k][j];
```

3.3 Statistics

The following table records the miss rate for running cache statistics these approach.

Miss Rate For matrix size 200

Cache config	Original	Transpose	Reordering	Blocking
16KB 32b 1	0.1081	0.0691	0.0341	0.0065
16KB 32b 2	0.0655	0.0635	0.0316	0.0049
16KB 32b 4	0.063	0.0633	0.0316	0.0047
16KB 64b 1	0.4311	0.0406	0.0188	0.0072
16KB 64b 2	0.1076	0.0321	0.0159	0.0051
16KB 64b 4	0.0732	0.0318	0.0158	0.0051
32KB 32b 1	0.0855	0.065	0.0328	0.0053
32KB 32b 2	0.0634	0.0633	0.0316	0.0045
32KB 32b 4	0.063	0.0633	0.0316	0.0042
32KB 64b 1	0.4206	0.0344	0.0174	0.0058
32KB 64b 2	0.0767	0.0318	0.0158	0.0041
32KB 64b 4	0.0315	0.0317	0.0158	0.0023

Improvement in miss rate for matrix size 200 over Original

Cache config	Transpose	Reordering	Blocking
16KB 32b 1	36%	68%	94%
16KB 32b 2	3%	52%	93%
16KB 32b 4	0%	50%	93%
16KB 64b 1	91%	96%	98%
16KB 64b 2	70%	85%	95%
16KB 64b 4	57%	78%	93%
32KB 32b 1	24%	62%	94%
32KB 32b 2	0%	50%	93%
32KB 32b 4	0%	50%	93%
32KB 64b 1	92%	96%	99%
32KB 64b 2	59%	79%	95%
32KB 64b 4	-1%	50%	93%
Average	36%	68%	94%

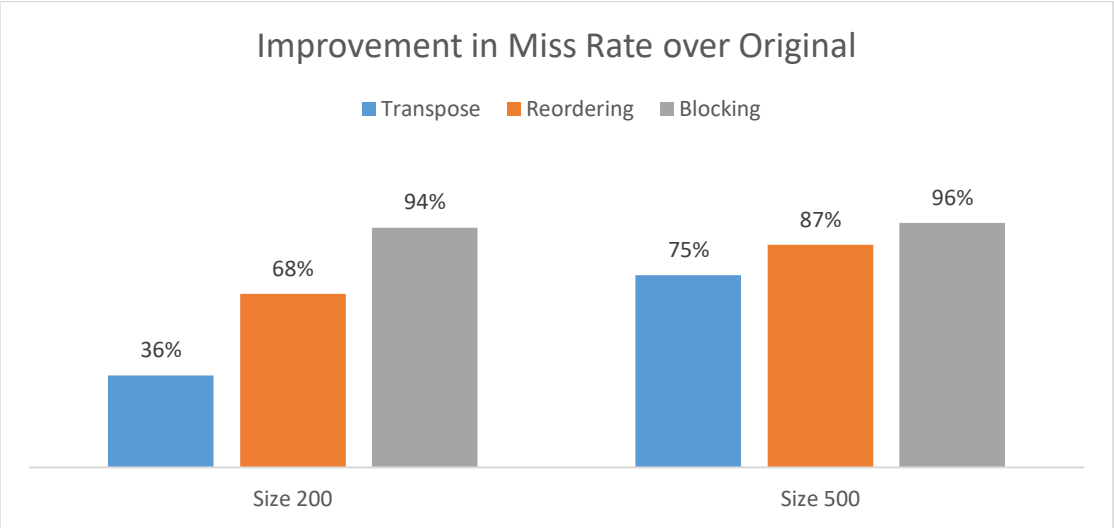
Miss Rate for matrix size 500

Cache config	Original	Transpose	Reordering	Blocking
16KB 32b 1	0.4032	0.0727	0.0399	0.015
16KB 32b 2	0.2333	0.0632	0.0315	0.01
16KB 32b 4	0.2958	0.0632	0.0314	0.0099
16KB 64b 1	0.5019	0.0405	0.0279	0.0188
16KB 64b 2	0.5313	0.0325	0.0158	0.0071
16KB 64b 4	0.5325	0.0323	0.0157	0.007
32KB 32b 1	0.3721	0.0683	0.0374	0.0124
32KB 32b 2	0.1234	0.063	0.0314	0.008
32KB 32b 4	0.0627	0.0628	0.0314	0.0077
32KB 64b 1	0.4438	0.0364	0.0253	0.0157
32KB 64b 2	0.2124	0.0318	0.0158	0.007
32KB 64b 4	0.177	0.0316	0.0157	0.007

Improvement in Miss rate for matrix size 500 over Original

Cache config	Transpose	Reordering	Blocking
16KB 32b 1	82%	90%	96%
16KB 32b 2	73%	86%	96%
16KB 32b 4	79%	89%	97%
16KB 64b 1	92%	94%	96%
16KB 64b 2	94%	97%	99%
16KB 64b 4	94%	97%	99%
32KB 32b 1	82%	90%	97%
32KB 32b 2	49%	75%	94%
32KB 32b 4	0%	50%	88%
32KB 64b 1	92%	94%	96%
32KB 64b 2	85%	93%	97%
32KB 64b 4	82%	91%	96%
Average	75%	87%	96%

3.4 Observations & Discussions



Following points were observed

Observation 1: **Order of improvement in miss rate is Transpose < Reordering < Blocking**

Explanation: It is due to the fact that for transpose and reordering essentially do the same thing, but for Transpose we need to also do copy operation, which increases miss rate and hence decreases improvement over original. Blocking is fastest as it exploits temporal locality at its maximum while Transpose, Reordering just exploit spatial locality. Hence number of expected misses is least in Blocking.

Observation 2: **Order of improvement in miss rate increases with increasing matrix size**

Explanation: This is inline with the general trend as size of matrix increases the order of improvement is bound to increase since accessing to cache increase. The order of improvement is a function of size of improvement matrix.

4 Conclusions

The following summaries the discussion

1. Increasing Block size decreases compulsory misses
2. Increasing Block size decreases capacity misses for smaller size matrix but increases capacity misses for larger size matrix
3. Increasing cache size decreases capacity misses
4. Increasing associativity decreases conflict misses
5. Hit rate is directly related to cache size and associativity
6. Hit rate is directly related to block size for smaller matrix but inversely for larger size matrix
7. The best possible configuration for 200 size matrix is 32KB cache, 64 bytes Block Size and associativity 4 while for 500 size matrix is 32KB cache, 32 bytes Block Size and associativity 4
8. Order of improvement in miss rate is Transpose < Reordering < Blocking
9. Order of improvement in miss rate increases with increasing matrix size

5 References

[1] “Computer Organisation and Architecture”, Dr. Smruti Ranjan Sarangi
[2] http://pages.cs.wisc.edu/~markhill/DineroIV/
[3] “What Every Programmer Should Know About Memory”, Ulrich Drepper, Red Hat, Inc. https://www.akkadia.org/drepper/cpumemory.pdf
[4] https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm
[5] http://functionspace.com/articles/40/Cache-aware-Matrix-Multiplication---Naive-isn--039;t-that-bad-