# Computer Architecture - Project 1 Analysis

Kevin Jun

February 3, 2020

## 2.1 - Correctness

Daxpy Vector Solution:

$$\{0.0, 5.0, 10.0, 15.0, 20.0, 25.0, 30.0, 35.0, 50.0\}$$

Block MxM ($f = 3$) Solution:

$$\begin{bmatrix} 3672 & 3744 & 3816 & 3888 & 3960 & 4032 & 4104 & 4176 & 4248 \\ 9504 & 9738 & 9972 & 10206 & 10440 & 10674 & 10908 & 11142 & 11376 \\ 15336 & 15732 & 16128 & 16524 & 16920 & 17316 & 17712 & 18108 & 18504 \\ 21168 & 21726 & 22284 & 22842 & 23400 & 23958 & 24516 & 25074 & 25632 \\ 27000 & 27720 & 28440 & 29160 & 29880 & 30600 & 31320 & 32040 & 32760 \\ 32832 & 33714 & 34596 & 35478 & 36360 & 37242 & 38124 & 39006 & 39888 \\ 38664 & 39708 & 40752 & 41796 & 42840 & 43884 & 44928 & 45972 & 47016 \\ 44496 & 45702 & 46908 & 48114 & 49320 & 50526 & 51732 & 52938 & 54144 \\ 50328 & 51696 & 53064 & 54432 & 55800 & 57168 & 58536 & 59904 & 61272 \end{bmatrix}$$

## 2.2 - Associativity

Judging from the results in the Table 1, this seems to be a reasonable design decision. There seems to be a clear plateau in miss rates even past a 4-way set associative cache. The results suggest that a 4-way set associative cache would perform just as well, but there must be some other benefit not seen in these results that would warrant using an 8-way set associative cache instead of just a 4-way.

| Cache Associativity | Instructions | Read Hits | Read Misses | Read Miss % | Write Hits | Write Misses | Write Miss % |
|---|---|---|---|---|---|---|---|
| 1 | 449971200 | 222219123 | 2420877 | 1.07767% | 3630720 | 516480 | 12.45370% |
| 2 | 449971200 | 223698840 | 941160 | 0.41896% | 4060800 | 86400 | 2.08333% |
| 4 | 449971200 | 223747200 | 892800 | 0.39744% | 4060800 | 86400 | 2.08333% |
| 8 | 449971200 | 223747200 | 892800 | 0.39744% | 4060800 | 86400 | 2.08333% |
| 16 | 449971200 | 223747200 | 892800 | 0.39744% | 4060800 | 86400 | 2.08333% |
| 1024 | 449971200 | 223747200 | 892800 | 0.39744% | 4060800 | 86400 | 2.08333% |

Table 1: Associativity

## 2.3 - Memory Block Size

The miss rates in Table 2 trend downwards where a minimum is reached at a block size of 256 bytes. However, for block sizes 512 and 1024, the miss rates steeply increase. Although, generally, larger blocks would mean a greater exploitation of spatial locality to lower miss rates, utilizing block sizes that constitute a significant portion of the total cache size (65,536 bytes for this analysis) can actually increase the miss rate. This is because the number of blocks that can be held in cache will decrease, causing greater competition for those limited block spots in cache. As a result, a block may be evicted from the cache before many of its words are accessed, increasing the miss rate.

| Block Size | Instructions | Read Hits | Read Misses | Read Miss % | Write Hits | Write Misses | Write Miss % |
|---|---|---|---|---|---|---|---|
| 8 | 449971200 | 217110720 | 7529280 | 3.35171% | 3456000 | 691200 | 16.66667% |
| 16 | 449971200 | 220875360 | 3764640 | 1.67585% | 3801600 | 345600 | 8.33333% |
| 32 | 449971200 | 222757680 | 1882320 | 0.83793% | 3974400 | 172800 | 4.16667% |
| 64 | 449971200 | 223698840 | 941160 | 0.41896% | 4060800 | 86400 | 2.08333% |
| 128 | 449971200 | 224169420 | 470580 | 0.20948% | 4104000 | 43200 | 1.04167% |
| 256 | 449971200 | 224404710 | 235290 | 0.10474% | 4125600 | 21600 | 0.52083% |
| 512 | 449971200 | 222725347 | 1914653 | 0.85232% | 3734992 | 412208 | 9.93943% |
| 1024 | 449971200 | 219351857 | 5288143 | 2.35405% | 2932456 | 1214744 | 29.29070% |

Table 2: Memory Block Size

## 2.4 - Total Cache Size

The minimum cache size required to achieve a 0.5% data read miss rate or less for the mxm_blocked algorithm using default settings is 65,536 bytes (64KB). The 32KB cache size with default settings has a 0.587% read miss rate. However, if we run the mxm_block algo with a 32KB cache set to 8-way set associativity as we were told the Skylake architecture typically is, we achieve a read miss rate of 0.486% which fulfills the 0.5% data miss rate target.

| Cache Size | Instructions | Read Hits | Read Misses | Read Miss % | Write Hits | Write Misses | Write Miss % |
|---|---|---|---|---|---|---|---|
| 4096 | 449971200 | 109688400 | 114951600 | 51.17147% | 0 | 4147200 | 100.00000% |
| 8192 | 449971200 | 206585790 | 18054210 | 8.03695% | 0 | 4147200 | 100.00000% |
| 16384 | 449971200 | 222907245 | 1732755 | 0.77135% | 3225600 | 921600 | 22.22222% |
| 32768 (default 2-way) | 449971200 | 223321204 | 1318796 | 0.58707% | 3946112 | 201088 | 4.84877% |
| 32768 (8-way) | 449971200 | 223547280 | 1092720 | 0.48643% | 4059120 | 88080 | 2.12384% |
| 65536 | 449971200 | 223698840 | 941160 | 0.41896% | 4060800 | 86400 | 2.08333% |
| 131072 | 449971200 | 223770827 | 869173 | 0.38692% | 4060800 | 86400 | 2.08333% |
| 262144 | 449971200 | 224077096 | 562904 | 0.25058% | 4060800 | 86400 | 2.08333% |
| 524288 | 449971200 | 224150552 | 489448 | 0.21788% | 4060800 | 86400 | 2.08333% |

Table 3: Total Cache Size

## 2.5 - Problem Size and Cache Thrashing

### Question 1

No, this is not the case for the regular matrix-matrix multiply algorithm. Matrices with $d = 480, 512$ exhibit significantly worse read and miss rates than the $d = 488$ problem size.

The poor performance in mxm for $d = 480, 512$ can be explained by examining the strides taken in each iteration through Matrix B - the matrix being iterated on the column. Strides are considered the number of bytes that must be traversed to reach the double at address $B[i][j]$ from $B[i-1][j]$ which can be calculated by multiplying $d$ by the size of a word (8 bytes). The table below shows the stride lengths for each $d$:

| $d$ | Stride Length (bytes) | Stride Length (words) |
|---|---|---|
| 480 | 3,840 bytes | 480 |
| 488 | 3,904 bytes | 488 |
| 512 | 4,096 bytes | 512 |

Stride Lengths

It is easy to see why a $d = 512$ would yield high miss rates because there are 512 sets in a two-way associative cache with $1,024$ blocks. In order to iterate through the necessary addresses in Matrix B, the algorithm must stride at a pace of 512 blocks or $4,096$ bytes per iteration. As a result, the same 8 sets are repeatedly accessed in the cache for each column, resulting in a large number of conflict misses and cache thrashing in which only a small portion of the cache is utilized and spatial locality is wasted. Because the same eight sets are repeatedly accessed, blocks are consistently evicted from the set and spatial locality cannot be utilized when iterating through the next column that would share the same block as the double directly to its left in the same row. Thankfully, the miss rate is approximately half because the addresses read in from Matrix A (which is iterated on the row) can be retained in cache to be used for the next $C[i][j]$ computation. Refer to the table below to see a visual representation of these strides:

| | | $j$ | 0 | $j$ | 8 |
|---|---|---|---|---|---|
| | | Address | Set Index | Address | Set Index |
| $i$ | 0 | 2097152 | 0 | 2097216 | 1 |
| | 1 | 2101248 | 64 | 2101312 | 65 |
| | 2 | 2105344 | 128 | 2105408 | 129 |
| | 3 | 2109440 | 192 | 2109504 | 193 |
| ... | ... | ... | ... | ... | ... |
| | 8 | 2129920 | 0 | 2129984 | 1 |
| | 9 | 2134016 | 64 | 2134080 | 65 |

Issues with addressing and strides

The first column is that row at which we are accessing into Matrix B and the values following $j$ are the columns that we are accessing. For example, the double at $B[0][0]$ is stored at Address 2097152 which corresponds to Set 0, and the double at $B[0][1]$ is stored at Address 2097216 which corresponds to Set 1. Moving down the columns, which is the same way that we would access the doubles of Matrix B, we see that the Set Index grows by 64 - the accessed set "strides" at a rate of 64 sets per row. Eventually, because there are only 512 sets in our 512x512 otherwise default-settings cache, this set value must repeat every 8 rows which is shown when we get to $B[8][0]$ and $B[8][1]$. The write miss rate is at 100% because the block associated with the address at $C[i][j-1]$ is definitely evicted by the time the inner for loop executes again and the CPU attempts to write to the address $C[i][j]$. This is because the address of Matrix A will share the same Set Index as an address of Matric C with the same $i$ and $j$ coordinates.

The same is true for the 480x480 block. The Set Indices repeat every 128 rows, which also results in a large number of conflict

misses because only two blocks can be stored per set. Thus, every attempt to load a double associated with an address in Matrix B would result in either a compulsory (in the beginning) or a conflict miss - just as in the case for the 512x512 matrix.

Performance for the 488x488 matrix improves significantly because Set Indices are not repeated while striding down the columns of Matrix B. Thus, spatial locality can be used when pulling values associated with addresses in Matrix A and Matrix B. For Matrix A, spatial and temporal locality are utilized because we are iteratively loading doubles that have a high chance of being stored in the same block because they were initialized in row-major. For Matrix B, spatial locality is utilized because all 488 blocks associated with a full column can be loaded into the cache and utilized for the next $k + 1$ iteration of the inner for loop. This is also why there is a massive improvement in the write miss rate from both the $d = 480, 512$ matrix multiplications. The inner for loop iterates through all $k$ up to $d = 488$ and because the Set Indices do not repeat within the 488 iterations, the associated $C[i][j]$ could be retained in the cache if Matrices A and B did not evict the previous $C[i]][j-1]$ address, assuming both $C[i][j]$ and $C[i]][j-1]$ are in the same data block.

## Question 2

It is not successful for the 512x512 matrix in Table 4. The read miss rate is approximately the same, and the write miss rate remains at a whopping 100%. Although the block multiplication method is meant to improve on the regular method, it fails in the case of the 512x512 matrix for the same reason mentioned in Question 1. Using a blocking factor of 32, that means that there is a 32x32 block in Matrix B that is accessed repeatedly in the matrix multiplication algorithm. This creates the same problem seen above in which the strides taken by the algorithm match up too perfectly with the dimensions of the matrix and cause continuous conflict misses. Because there are 512 sets in the default settings of the cache, the same 8 sets will be repeatedly accessed. Thus, nothing has really changed by setting a blocking factor of 32 because the blocking factor is larger than the repeat rate of the sets (8).

The write miss rate skyrockets to 100% because although it is loaded to start the calculation of cell $C[i][j]$, it will always get evicted when $A[i][k]$ and $B[k][j]$ are loaded because it will share a set with both a value of $A[i][k]$ and $B[k][j]$ as Matrices A and B are accessed in the innermost loop. The indices of both Matrix A and Matrix B will match the $i, j$ indices of Matrix C at some point in this iteration and because these matrices are of dimension 512x512 with 512 sets, the address of matching $i, j$ index words for Matrices A, B, and C map into the same set in cache.

## Question 3

There is significant improvement as the associativity increases because there are less collisions as the size of the set increases. Naturally, as the size of a set increases, there will be less fighting for positions within the set as blocks are loaded into the cache and there is also a greater chance that the block that we want to access is within the set.

This implementation of a cache searches sequentially within the set to see if the desired block is already loaded. However, the real implementation of a cache has $n$ comparators given an $n$-way associative cache so that all $n$ entries in a given set can be searched at once. Implementing a fully associative cache would require a comparator per entry, making it incredibly expensive and difficult to deploy. While increasing associativity does decrease miss rates, it does so with diminishing returns with a possible increase access times.

## Question 4

The cache performance given a block matrix multiplication algorithm of matrices of size 512 x 512 and an 8-way set associative cache could be optimized by using a blocking factor less than 16 and/or loading the B matrix such that the columns utilize spatial locality instead of the rows.

**Decreasing the blocking factor**

Decreasing the blocking factor could decrease miss rates because the number of words fighting for spots in any given set would decrease even when taking into account that the strides taken through Matrix $B$ result in the same sets accessed over and over again. Because conflict misses are guaranteed in the matrix multiplication of $n$ x $n$ matrices where $n$ is a power of 2, decreasing the blocking factor would at least try to minimize the number of times that a conflict miss occurs.

The table below shows the cache performance using different blocking factors.

| Blocking factor | Read Miss Rate | Write Miss Rate |
|---|---|---|
| 1 | 37.56510% | 0.07282% |
| 2 | 12.53906% | 0.14479% |
| 4 | 4.18837% | 0.28626% |
| 8 | 2.02776% | 0.55970% |
| 16 | 28.03030% | 92.50000% |
| 32 | 49.71154% | 86.18421% |

Effect of blocking factor on 512x512 matrix with 8-way set associative cache

It seems that a blocking factor of 4 or 8 might be the sweet spot of getting the most optimal read/write miss rates. Given there are an incredibly high number of read requests sent to the cache in this algorithm, the blocking factor of 8 may be better, but there may be benefits to the blocking factor of 4 because there are also a high number of write requests in the block matrix multiplication algorithm. This makes sense taking into account what was discussed in Question 1 when we found that the 512 x 512 matrix multiplication repeats Set Indices every 8 iterations through Matrix B. Decreasing the blocking factor to be at or below the 8 repeat rate would intuitively decrease the number of conflict misses - something that the blocking factor of 32 couldn't achieve and was explained in Question 2.

**Column-wise loading**

Loading the data such that the columns share blocks would enforce some level of spatial locality in the same way that accessing Matrix A sequentially row-wise utilizes spatial locality. This would require setting up a different for loop when loading the data into Matrix B. For example, the following matrix would have Column A stored first, iterating downwards, and then Column B stored after Column A has been stored:

| $A$ | $B$ | $C$ | $D$ |
|---|---|---|---|
| 0 | 2 | 4 | 8 |
| 512 | 514 | 516 | 518 |
| 1024 | 1026 | 1028 | 1030 |
| 1536 | 1538 | 1540 | 1542 |

Because Matrix B must be accessed columnarly and we have stored the data in the same way, it would utilize spatial locality although the for loop would be more difficult to set up and perhaps the algorithm would be more difficult to understand without understanding the implications on cache performance of storing the matrix in this way.

## Tables for 2.5

| Matrix Dimension | MxM Method | Blocking Factor | Instructions | Read Hits | Read Misses | Read Miss % | Write Hits | Write Misses | Write Miss % |
|---|---|---|---|---|---|---|---|---|---|
| 480 | Regular | - | 443289600 | 107035281 | 114148719 | 51.60804% | 604800 | 316800 | 34.37500% |
| 480 | Blocked | 32 | 449971200 | 223698840 | 941160 | 0.41896% | 4060800 | 86400 | 2.08333% |
| 488 | Regular | - | 465809664 | 217464849 | 14963695 | 6.43798% | 833504 | 119072 | 12.50000% |
| 488 | Blocked | 8 | 494625088 | 244703648 | 2251680 | 0.91178% | 15151912 | 89304 | 0.58594% |
| 512 | Regular | - | 537919488 | 133891584 | 134543872 | 50.12150% | 0 | 1048576 | 100.00000% |
| 512 | Blocked | 32 | 546045952 | 137084928 | 135544832 | 49.71755% | 0 | 4980736 | 100.00000% |

Table 4: MM Problem - Associativity = 2

| Matrix Dimension | MxM Method | Blocking Factor | Instructions | Read Hits | Read Misses | Read Miss % | Write Hits | Write Misses | Write Miss % |
|---|---|---|---|---|---|---|---|---|---|
| 480 | Regular | - | 443289600 | 107118315 | 114065685 | 51.57050% | 604800 | 316800 | 34.37500% |
| 480 | Blocked | 32 | 449971200 | 223747200 | 892800 | 0.39744% | 4060800 | 86400 | 2.08333% |
| 488 | Regular | - | 465809664 | 217871996 | 14556548 | 6.26281% | 833504 | 119072 | 12.50000% |
| 488 | Blocked | 8 | 494625088 | 244447601 | 2507727 | 1.01546% | 15151912 | 89304 | 0.58594% |
| 512 | Regular | - | 537919488 | 133892096 | 134543360 | 50.12131% | 688128 | 360448 | 34.37500% |
| 512 | Blocked | 32 | 546045952 | 137101312 | 135528448 | 49.71154% | 688128 | 4292608 | 86.18421% |

Table 5: MM Problem - Associativity = 8

| Matrix Dimension | MxM Method | Blocking Factor | Instructions | Read Hits | Read Misses | Read Miss % | Write Hits | Write Misses | Write Miss % |
|---|---|---|---|---|---|---|---|---|---|
| 480 | Regular | - | 443289600 | 207331202 | 13852798 | 6.26302% | 806400 | 115200 | 12.50000% |
| 480 | Blocked | 32 | 449971200 | 223747200 | 892800 | 0.39744% | 4060800 | 86400 | 2.08333% |
| 488 | Regular | - | 465809664 | 217871994 | 14556550 | 6.26281% | 833504 | 119072 | 12.50000% |
| 488 | Blocked | 8 | 494625088 | 245079944 | 1875384 | 0.75940% | 15151912 | 89304 | 0.58594% |
| 512 | Regular | - | 537919488 | 251625474 | 16809982 | 6.26221% | 917504 | 131072 | 12.50000% |
| 512 | Blocked | 32 | 546045952 | 271548416 | 1081344 | 0.39663% | 4882432 | 98304 | 1.97368% |

Table 6: MM Problem - Fully Associative

## 2.6 - Replacement Policy

The LRU replacement policy performs the best with both the lowest read and write miss rates. FIFO is a close second and the Random replacement policy is a close third. It is unsurprising that the temporal-based replacement policies result in the best miss rates because they are able to utilize temporal locality. LRU's performance is dependent on the assumption that blocks that were most recently accessed will be soon accessed again which is helpful in the case of matrix multiplication which is a series of sequential accesses into the same blocks/sets. FIFO is similar in that the assumption is that a datablock that was first accessed a long time ago will most likely not be used as time goes on.

Although random doesn't utilize temporal or spatial locality, it could be a viable choice in scenarios where caches are significantly large and the chance of a miss increases. This is because LRU's performance depends on a loop or iteration that is relatively tight - that is, a loop that traverses the cache well such that each iteration of the loop results in the same indices/data blocks being accessed which happens when the size of the loop matches well with the size of the cache. The random replacement policy degrades well as cache size increases because the eviction policy is not associated with the cache size or the size of the iterations (for loop).

Furthermore, there is something to be said regarding the cost of implementing LRU over FIFO and random. As with all replacement policies, there is a trade-off between latency and hit rate. LRU has a good chance of having a better hit ratio because it stores a lot of information about the data block that was accessed (i.e. age bits that denote the access rank); however, this means that there can be higher latency because more data must be parsed to return the desired data block. FIFO has slightly less information needed to be stored and random has almost none, resulting in a higher miss rate but lower latency.

| Replacement Policy | Instructions | Read Hits | Read Misses | Read Miss % | Write Hits | Write Misses | Write Miss % |
|---|---|---|---|---|---|---|---|
| Random | 449971200 | 223627400 | 1012600 | 0.45077% | 4058820 | 88380 | 2.13108% |
| FIFO | 449971200 | 223629200 | 1010800 | 0.44996% | 4059340 | 87860 | 2.11854% |
| LRU | 449971200 | 223698840 | 941160 | 0.41896% | 4060800 | 86400 | 2.08333% |

Table 7: Replacement Policy