CS610 PROGRAMMING FOR PERFORMANCE
Assignment 1

August 26, 2024

SAHIL BASIA
241110061

Note

- Notation used:- Number of elements that can fit in 1 block = BL, 'N' is the variable used in question
- All the commands that are executed, are done in super-user(root) privileges

Ans: **Problem - 1**

Given:-
Size of the cache (C) = 256KB
Associativity used = 8-way set associative
Line size/Block size (BS) = 64B
Word size (W) = Size of each element (float data type)= 4B
Size (size variable given in question)= 32K = 2^{15}
Size of each element (float data type)= 4B

Array size = (No of elements in array) × (size of each element)= 32K * 4B = 128KB

Total no of lines in the cache = $\frac{\text{Cache size}}{\text{Line size}} = \frac{256KB}{64B} = 4K$

Number of sets in the cache = $\frac{\text{Number of the cache lines}}{\text{Cache set associativity}} = \frac{4K}{8} = 512$

Number of blocks for array = $\frac{\text{Number of elements in array}}{\text{No of elements in 1 block}} = \frac{32K}{16} = 2K$

Number of elements that can fit in 1 block of the cache = $\frac{\text{Block size}}{\text{Size of 1 element(size of float data type)}}$
$= \frac{64B}{4B} = 16$
Therefore we can say that whenever the first element of an array block is accessed and if that access leads to a miss, then that array block will be brought to the cache. However, along with that missed element inside that array block, the other 15 adjacent elements of the array will also be present inside the same block due to spatial locality. So when those other 15 elements are accessed they will result in a cache hit.

From the calculated data we can see the array size = $\frac{1}{2}$(the cache size). Therefore we can conclude that the whole array will completely fit inside the cache. Also, we can say that capacity and conflict misses will never occur, because the cache can easily accommodate the whole array so there will be no eviction of any of the cache blocks due to the small size of the cache, Also there will be no use of LRU replacement policy in this question.

We can further conclude that for only the first iterations we will be facing the cold misses and for the next further iterations we will always face the hits in the cache whenever the inner loop runs for that particular iteration and elements of array A are accessed. This is because the array will be completely filled in the cache in the first iteration only and none of the blocks is evicted due to the large cache size at any point of time during the iteration of the inner loop and outer loop.
So we will only consider the cache miss analysis for only the 1^{st} *iteration for every stride pattern asked.*

Stride-1 access
In stride-1 access, we will be accessing the elements of an array in a sequential pattern only with a jump of 1 from the previously accessed element. Now when 1^{st} element of the array block is accessed it will result in a cold miss in the cache. Therefore this memory block containing missed elements will be copied into the cache. But when the next 15 adjacent elements are accessed in the inner loop, they will result in a cache hit as they were also present in that same block that was copied to the cache on the cold miss of that element. In a more generalized way we can say that, a miss will occur in this pattern 0, 16, 32, ... so on.

Total the cache miss analysis:-
Number of cold misses in accessing first 16 elements by inner loop = 1
Therefore when all 32K elements are accessed, the number of cold misses = $32K \times \frac{1}{16}$ = 2K cold misses = 2^{11} cold misses
Capacity misses = 0
Conflict misses = 0

Stride-4 access
In stride-4 access, now the elements will be accessed by a jump of 4 from the previously accessed element, i.e. the access pattern will be 0, 4, 8, 12, 16, .. so on. But one block size can comprise 16 elements of an array, so there will be no misses for 4, 8, or 12 access to elements. The miss pattern will be 0, 16, 32 ... so on. This is because whenever the first miss occurs, the block of 16 elements with that missed element will be brought to the cache. Now when the next elements are accessed in

the stride 4 pattern, they will not result in a cold miss in the cache, until all the elements of the block are not finished.
In a more generalized way, we can say that every $16 \times i$ (where $i \in \{0, 1, 2, 3, 4...\}$)element of the array will result in the cache miss if that element is accessed.

Total the cache miss analysis:-
Number of cold misses for first 16 elements = 1
For 32K elements, number of cold misses = $32K \times \frac{1}{16}$ = 2K cold misses = 2^{11} cold misses
Capacity misses = 0
Conflict misses = 0

Stride-16 access:-
The access pattern will be 0, 16, 32, ... so on i.e., the access is done block by block. Now the miss pattern will also be the same, as discussed in stride-4 access.

Total the cache miss analysis:-
Number of cold misses for first 16 elements = 1
For 32K elements , number of cold misses = $32K \times \frac{1}{16}$ = 2K cold misses = 2^{11} cold misses
Capacity misses = 0
Conflict misses = 0
Alternative solution:- For this 16-stride access, the number of cold misses can also be calculated using no of blocks of the whole array. Because the access is block by block,
Number of cold misses for first block = 1
Therefore number of cold misses for 2K blocks = 2K cold misses = 2^{11} cold misses

Stride-64 access:-
The access pattern of elements will be 0, 64, 128 ... so on. Now the missing pattern will also be 0, 64, 128, ... so on. Because we are using stride 64 access, the elements will be accessed by a jump of 64 elements from the previous elements accessed. We can also say, that after accessing the 1^{st} element of 0^{th} block, we will be accessing 1^{st} element of 4^{th} block and further 1^{st} element of 8^{th} and so on. So we are basically jumping 4 blocks from the previous block accessed.

Total the cache miss analysis:-
For 64 elements, no of cold misses = 1
{Note: for elements from 0 to 63 only one element is accessed and that will be miss}
For 32K elements, no of cold misses = $\frac{32K}{64}$ = 512 cold misses

Capacity misses = 0
Conflict misses = 0
Alternative solution:- For this 16-stride access, the number of cold misses can also be calculated using no of blocks of the whole array. Because the access is block by block,
Number of blocks jumped = 4
Number of cold misses for first block = 1
Therefore number of cold misses for 2K blocks $=\frac{2K}{4} = 512$ cold misses

Stride-8K access:-
The access pattern of elements will be 0, 8k,16k ... so on. Because we are now accessing only every 8K $\times$ i (where $i \in \{0, 1, 2, 3, 4...\}$) element of the array, so only these will result in cold misses.

Total the cache miss analysis:-
For 8K elements, the number of cold misses = 1
For 32K elements, number of cold misses = $\frac{32K}{8K} = 4$ cold misses
Capacity misses = 0
Conflict misses = 0

Stride-16K access:-
The access pattern will be 0, 16K. For this stride, only 2 accesses will be done on the array.

Total the cache miss analysis:-
For 16K elements, the number of cold misses = 1
For 32K elements, number of cold misses = $\frac{32K}{8K} = 2$ cold misses
Capacity misses = 0
Conflict misses = 0

Stride-32k access:-
Only one time, the array will be accessed, i.e. for 0^{th} element of the array.

Total the cache miss analysis:-
The number of cold misses = 1
Capacity misses = 0
Conflict misses = 0
{Note:- For stride-8, 16K, 32K access we can also find cold miss using that alternative solution approach also}

Stride	Cold Misses	Capacity Misses	Conflict Misses
1	2^{11}	0	0
4	2^{11}	0	0
16	2^{11}	0	0
64	512	0	0
8K	4	0	0
16K	2	0	0
32K	1	0	0

Ans: **Problem - 2**

Given:-
Cache Size = 64K words= 2^{16} words
Line size/ Block size = 8 words
Number of elements that can fit in 1 block (**BL**) = 8
Number of the cache blocks/line = $\frac{2^{16}}{8} = 2^{13}$ blocks
Dimensions of matrix = 1024×1024
Size of each element of matrix = 1 word
Total size of matrix = $(1024 \times 1024) \times 1word = 2^{20}words$
Number of blocks in matrix = $\frac{2^{20}words}{8\ words} = 2^{17}blocks$
N variable = 1024

From this calculated information we can infer that the size of the matrix is large, as compared to the size of the cache. Therefore whole array will never fit inside the cache completely.
Amount of matrix that can fit in the cache = $\frac{CacheSize}{MatrixSize} = \frac{2^{16}}{2^{20}} = \frac{1}{2^4}$
So only $\frac{1}{16}$ of the matrix will fit inside the cache.

For direct-map, the cache configuration, the blocks of the matrix will be mapped according to the mapping function of direct-map configuration. Since arrays are stored in row-major order, there will be 2^7 blocks in 1 row of the matrix (elements in the matrix are stored in row-major order, so a block of elements formed considers the same row-major) and total 2^{10} rows of these 2^7 blocks. Now blocks of 1^{st} row will map to the cache in a contiguous pattern from 0^{th} to $(2^7 - 1)^{th}$ blocks of the cache. So when a block of the next row is accessed it will map to 2^{7th} block of the cache and this continues. The mapping of matrix blocks to the cache blocks can be generalized using the formula [**(matrix block number)%(total the cache blocks)**]. For 1^{st} row of the matrix, the numbering of blocks will be from 0 to $2^7 - 1$. Now for the next row, the numbering of blocks will be from 2^7 to $2^{14} - 1$. Similarly, for other

rows, the numbering continues. But the cache has only 2^{13} blocks, so blocks of matrix from 0 to $2^{13} - 1$ will completely fill the cache. But when matrix block number >= 2^{13} of matrix is accessed it will wrap around because of the modulus operator in the formula and will start mapping from 0^{th} block of the cache again, thus evicting the older blocks. In row-major access, the matrix blocks accessed are numbered 0, 1, 2 .. so on. But in column-major access, the matrix blocks accessed are numbered in $0, 2^7, 2^8, 2^8 + 2^7$, .. so on, i.e. in the gap of 2^7 blocks.
For fully associative-map the cache configuration, the blocks of the matrix can map to any of the blocks of the cache. So there will be no need for a mapping function here. So 1 block of matrix can map to any of 2^{13} blocks of the cache. Later when all blocks of the cache are filled up, then eviction of blocks will occur if a block of matrix accessed is a miss.

Part-1 kij form

```
for (k = 0; k < N; k++){   // Loop 3
  for (i = 0; i < N; i++){   // Loop 2
    for (j = 0; j < N; j++){   // Loop 1
      C[i][j]+= A[i][k] * B[k][j];
    }
  }
}
```

For A

- **Now the loop 1, j loops from 0 to N-1**. A is not dependent on the j variable. When initially j=0 iteration runs keeping i,k fixed, the element $A[i][k]$ {where i and k are from outer loops} will result in a miss, and therefore the block containing that element will be brought to the cache. Now for the next iterations of j, $A[i][k]$ will always have hit in the cache due to temporal reuse, because for all 'j' same $A[i][k]$ is accessed. Therefore for the j loop, there will be **1** miss. This will be the same for both direct and fully associative cache.
- **Now for loop 2, the i loops to 0 to N-1**. The elements of $A[i][k]$ will be accessed in column-major order when i is iterated keeping j and k fixed. Now for each iteration of i, we will access the element in the particular column, and each element will give a miss. 1 column has N elements. So there will be **N** misses for this loop for both direct-mapped and fully associative the cache.
- **Now for loop 3, k loops from 0 to N-1**.When k is iterated keeping i, j fixed, the elements of A will be accessed in row major order. In the case of direct-mapped

configuration, there will be **N** misses. This is because, for k=0, when inner-loop i loops from 0 to N-1, all the elements are accessed in column-major order. Now, the blocks of the matrix will be the cached using mapping function. When iterations of i are over, the cache will contain the last accessed blocks of the matrix and these blocks have evicted some of the older blocks as the cache size is smaller than the matrix size. So now for k=1, when elements of the next columns of the matrix are accessed, they will not be present in the cache, therefore leading to miss. So for loop k, there will be **N** miss as for each iteration of k, this pattern will be repeated. In case of fully associative configuration, the loop k will have $\frac{\mathbf{N}}{\mathbf{BL}}$ miss. This is because, for k=0, when i iterate completely, it will cache the blocks in the cache without any eviction, as the cache has 2^{13} block, and in 1 column of the matrix, there are 2^{10} blocks. So the cache will not be filled completely with these 2^{10} blocks when the i loop completes and therefore no need to evict any block. So these 2^{10} blocks will map anywhere in the cache, unlike in direct-mapped cache the blocks have to be placed in their calculated block location only. So when the k=1 iteration starts, the blocks that are already in the cache will contain the elements because of spatial locality.

For B

- **Now the loop** 1**, j loops from 0 to N-1**. The elements of B are accessed in row major order when j is iterated keeping i, k fixed. So due to spatial locality and row-major access of elements, for loop j, there will be $\frac{\mathbf{N}}{\mathbf{BL}}$ misses, because for only the first element of block, it will be missed and for rest elements in the same block it will be hit. This will be the same for both direct and fully associative cache .
- **Now the loop** 2**, i loops from 0 to N-1**. The elements of B are not indexed using the i variable. When i=0 runs completely, all the elements of B[k][j] for k=0 and j=0 to j=N-1(just one whole 1 row of the matrix) are cached in the cache as initially they are not present in the cache, so they will give miss. The matrix blocks for the whole row of elements will map from block 0 to block $2^7 - 1$ in the cache. So when the next iteration of i starts from i=1, all access to elements of matrix B for the same k=0 and j=0 to j=N-1 will be hit. Now due to temporal reuse by other iterations of i, there will be only **1** miss for this loop.
- **Now the loop** 3**, k loops from 0 to N-1**. The elements are accessed in column major order when k is iterated keeping i, j fixed. In both direct-mapped and fully associative cache, there will be **N** misses because while traversing a particular column there will be N misses, as each column has N elements.

For C

- **Now the loop** 1**, j loops from 0 to N-1**. The access pattern of elements of

C is in row-major order when j is iterated, keeping i, k fixed. So due to spatial locality and row-major access, loop j will have $\frac{N}{BL}$ misses. For both direct and fully associative mapping it will be the same.

- **Now the loop 2, i loops from 0 to N-1**. For this loop, their miss will be **N** for both direct and fully associative mapping. This is because of column-major access of elements of C when i is iterated, keeping j, k fixed. Every access to an element of that particular column will give miss.

- **Now the loop 3, k loops from 0 to N-1**. For this loop, there will be **N** misses for both direct and fully associative cache configuration. This is because, for k=0, the whole matrix of C will be accessed. Since the number of blocks of the matrix is greater than the number of blocks of the cache, so at a particular instance of time the cache will be completely filled by some of the matrix blocks, and the leftover blocks will then start evicting the older blocks. So when the k=1 iteration starts, 1^{st} element of the matrix is accessed again, and it will give a miss in the cache, as the block containing that element had been evicted in the k=0 loop. This will be repeated for each iteration of k i.e. N times. So there will be N misses.

Table-2.1 - Summary of the cache miss analysis for **direct-map the cache configuration for kij**

	A	**B**	**C**
k	N	N	N
i	N	1	N
j	1	$\frac{N}{BL}$	$\frac{N}{BL}$
TM	$N^2 = 1024^2 = 2^{20}$	$\frac{N^2}{BL} = \frac{1024^2}{8} = 2^{17}$	$\frac{N^3}{BL} = \frac{1024^3}{8} = 2^{27}$

Table-2.2 - Summary of the cache miss analysis for **fully associative mapping configuration kij**

	A	**B**	**C**
k	$\frac{N}{BL}$	N	N
i	N	1	N
j	1	$\frac{N}{BL}$	$\frac{N}{BL}$
TM	$\frac{N^2}{BL} = \frac{1024^2}{8} = 2^{17}$	$\frac{N^2}{BL} = \frac{1024^2}{8} = 2^{17}$	$\frac{N^3}{BL} = \frac{1024^3}{8} = 2^{27}$

Part-2 jik form

```
for (j = 0; j < N; j++){   // Loop 3
  for (i = 0; i < N; i++){   // Loop 2
    for (k = 0; k < N; k++){   // Loop 1
      C[i][j]+ = A[i][k] * B[k][j];
    }
  }
}
```

For A

- **Now the loop 1, k loops from 0 to N-1**.The elements of A will be accessed in row-major order when k is iterated, keeping i, j fixed. So the miss will be $\frac{\mathbf{N}}{\mathbf{BL}}$ for both directed and fully associative cache configuration, due to spatial locality.
- **Now the loop 2, i loops from 0 to N-1**. The elements are accessed in column-major access when i is iterated keeping j, k fixed. For this loop, there will be **N** misses for both directed and fully associative cache configuration. Because in each column N distinct element will be accessed, so N misses.
- **Now the loop 3, j loops from 0 to N-1**. The $A[i][k]$ is indexed on i and k variables only. So for loop j=0, the whole matrix of A will be traversed. Because the size of the cache is smaller than the size of the matrix, some blocks of the matrix will evict the blocks of the cache and the cache will contain the most recent access of blocks of the matrix. So when the j=1 iteration starts, the elements of the matrix starting from k=0,i=0 will give a miss, as they were evicted by other blocks. So for j=1, the same miss pattern as j=0 will be repeated and this will be done for N times, as j iterates from 0 to N-1. This miss pattern will happen in both direct and fully associative the cache. So there will be **N** misses for this loop, for both direct and fully associative the cache.

For B

- **Now the loop 1, k loops from 0 to N-1**. The elements of B will be accessed in column-major when k is iterated, keeping i, and j fixed. For both direct and fully associative cache, there will be **N** misses. This is because when a particular column of a matrix is traversed, it contains N elements and those elements do not share any spatial locality, thus giving N miss.
- **Now the loop 2, i loops from 0 to N-1**. Matrix B is not indexed on variable i. When the i=0 loop ends, only the block containing the element of a single column

(inner loop k) is cached. In the case of direct-map the cache, the mapping is done on the basis of the mapping function, therefore the blocks will be mapped in the gap of 2^7. However, due to the small size of the cache, some blocks of the matrix can be wrapped up to the starting block of the cache because of the mapping function thus evicting those older blocks. So when loop i=1 starts again, the starting blocks of the same columns will give a miss in the cache, therefore for loop i this miss pattern will repeat N times, so **N** misses. In fully associative the cache, the blocks of the matrix can be mapped to any of the blocks of the cache. Now for i=0, all blocks of a particular column will be cached in the cache. Because there is a total of 2^{13} blocks in the cache, N=2^{10} matrix blocks that are traversed in column-major will map anywhere in the cache and there will be no eviction as the cache will still remain empty. Now when the i=1 iteration starts, the same column will be accessed and all the elements will give a hit. So for this loop only **1** miss.

- **Now the loop 3, j loops from 0 to N-1**. The elements of matrix B will be accessed in row-major order when j is iterated, keeping i, k fixed. In fully associative the cache, their will be $\frac{\mathbf{N}}{\mathbf{BL}}$ misses. This is because, for j=0 iteration, inner loop k=0 to k=N-1 runs, and the elements of the B matrix are accessed in column-major order. Initially, the elements will not be in the cache, so they will give miss and blocks of those missed elements will be cached in the cache. The blocks can map anywhere in the cache and only 2^{10} blocks are accessed in 1 column major access. So these 2^{10} blocks of the matrix can be mapped anywhere in 2^{13} blocks of the cache, without any eviction. Now when the j=1 iteration starts, the k loops from 0 to N-1 again. But now due to spatial locality the elements now accessed for j=1 columns will give a hit. For direct-mapped the cache, there will be **N** misses.

For C

- **Now the loop 1, k loops from 0 to N-1**. The matrix $C[i][j]$ does not use the k variable for indexing. For both direct-map the cache configuration and fully associative the cache configuration there will be **1** misses because of temporal reuse of the last accessed element of the matrix.
- **Now the loop 2, i loops from 0 to N-1**. The elements for matrix $C[i][j]$ will be accessed in column-major order when i is iterated, keeping k, j fixed. For direct-map the cache and fully associative the cache, both will have the same miss i.e. **N**. This is because, when elements in a particular column are accessed, initially they will not present in the cache, so they will result in a miss. 1 column of the matrix has N elements.
- **Now the loop 3, j loops from 0 to N-1**. For this, elements are accessed in row-major access when j is iterated, keeping i,k fixed. So for direct-mapped the cache, there will be **N** misses. For fully associative the cache it will be $\frac{N}{BL}$ because of spa-

tial locality, the adjacent elements are already cached along with the missed element block.

Table-2.3 - Summary of the cache miss analysis for **direct-map the cache configuration for jik**

	A	**B**	**C**
j	N	N	N
i	N	N	N
k	$\frac{N}{BL}$	N	1
TM	$\frac{N^3}{BL} = \frac{1024^3}{8} = 2^{27}$	$N^3 = 2^{30}$	$N^2 = 2^{20}$

Table-2.4 - Summary of the cache miss analysis for **fully associative mapping configuration for jik**

	A	**B**	**C**
j	N	$\frac{N}{BL}$	$\frac{N}{BL}$
i	N	1	N
k	$\frac{N}{BL}$	N	1
TM	$\frac{N^3}{BL} = \frac{1024^3}{8} = 2^{27}$	$\frac{N^2}{BL} = \frac{1024^2}{8} = 2^{17}$	$\frac{N^2}{BL} = \frac{1024^2}{8} = 2^{17}$

Ans: **Problem - 3**

```
#define N (4096)
double y[N], X[N][N], A[N][N];
for (k = 0; k < N; k++){   // Loop 3
  for (j = 0; j < N; j++){   // Loop 2
    for (i = 0; i < N; i++){   // Loop 1
      y[i] = y[i] + A[i][j] * X[k][j];
    }
  }
}
```

Given:
1 element size = 1 word
Cache size = 16MB = 2^{24}B
Block/line size = 32B = 2^5B
1 Word size = 8B = 2^3B
N variable = 4096 = 2^{12}
Number of words that can fit in 1 block (**BL**)= $\frac{\text{Block size}}{\text{1 Word size}} = \frac{2^5}{2^3} = 2^2 = 4$
Total number of the cache blocks = $\frac{\text{Cache size}}{\text{1 Block size}} = \frac{2^{24}}{2^5} = 2^{19}$
Cache configuration used = Direct-mapped the cache

Size of array y = $2^{12} \times 2^3 B = 2^{15} B$
Total blocks in array y = $\frac{\text{Array size}}{\text{1 Block size}} = \frac{2^{15}}{2^5} = 2^{10}$
Size of matrix X = $2^{12} \times 2^{12} \times 2^3 B = 2^{27} B$
Size of matrix A = $2^{12} \times 2^{12} \times 2^3 B = 2^{27} B$
Total blocks in matrix = $\frac{\text{Matrix size}}{\text{1 Block size}} = \frac{2^{27}}{2^5} = 2^{22}$

From this information, we can say that array y will completely fit inside the cache. Matrix A and X will not completely fit inside the cache due to the small size of the cache.

Mapping function :-[**(matrix/array block number)**%**(total the cache blocks)**]

For array y

- **For loop 1**, i iterates from 0 to 4095. When y[0] is accessed, it will result in a cache miss. So the block containing y[0] will move to the cache. Along with this missed element, 3 more elements of the array adjacent to the missed element will move inside this same block due to spatial locality. Because the cache used is a direct-mapped cache, the block will be mapped using a mapping function. So it will map to 0^{th} block of the cache. Now for the next 3 elements of array y, it will be hit. So for i, miss will be $\frac{\mathbf{N}}{\mathbf{BL}}$.
- **For loop 2**, j iterates from 0 to 4095. For loop j, a miss will be **1**. This is because for j=0, loop 1 runs and caches every element of array y in the cache. So when loop j=0 ends, the array y is completely stored in the cache. Thus on the next iteration of j, the temporal reuse of the same elements will happen and will give hits in the cache.
- **For loop 3**, k iterates from 0 to 4095. For this, the miss will be **1**. Same reason as above, due to temporal reuse of the cached elements of array y.

For matrix A

- **For loop 1**. Matrix A is accessed in column-major order when i is iterated, keeping values of j, and k fixed. Each each iteration i, N elements will be accessed. Therefore for this loop, it will be **N** misses.
- **For loop 2**. Matrix A is accessed in row-major order when j is iterated, keeping i,k fixed. The misses for this loop will be **N**, because when j=0 loop ends, because of the inner loop i, the last accessed blocks will evict the older blocks as these blocks are wrapped up to the starting block of the cache, due to small size of the cache and mapping function of direct-mapped the cache.
- **For loop 3**. Matrix indices are not dependent on variable k. So if we keep i, j fixed, the miss for this loop will **N**, as for each k, there will be no temporal reuse of all the elements accessed for k=0. So for each k, the matrix is accessed again, without any temporal reuse.

For matrix X

- **For loop 1**. The matrix is not indexed on the i variable. Keeping the j, k variable fixed when the iteration of i starts, for only i=0, the miss will occur, then for the rest of iterations of i, it will be hit, because of the temporal reuse of element accessed when i was 0. So for this loop only **1** miss will occur.
- **For loop 2**. The matrix will be accessed in row-major order when j is iterated, keeping i, k fixed. Their will be $\frac{\mathbf{N}}{\mathbf{BL}}$ misses. The reason is because of spatial locality, after one miss, the adjacent elements accessed in row-major order for that particular block will give a hit. So for each block of BL elements, we are getting 1 miss. So for N elements, it will be $\frac{N}{BL}$ misses.
- **For loop 3**. The matrix will be accessed in column-major order when k is iterated, keeping i, j fixed. Their miss will be **N** misses, because the elements accessed in this order will not share any spatial locality, with the previously accessed elements that gave miss. So each element accessed in a particular column will give a miss.

Table-3.1 -the cache miss analysis and total miss data

	y	**A**	**X**
k	1	N	N
j	1	N	$\frac{N}{BL}$
i	$\frac{N}{BL}$	N	1
TM	$\frac{N}{BL} = \frac{4096}{4} = 2^{10}$	$N^3 = 4096^3 = 2^{36}$	$\frac{N^2}{BL} = \frac{4096^2}{4} = 2^{22}$

Ans: **Problem - 4**

Part i)

The first 3 columns of the table, denote the blocking values used for each of the matrices A, B, and C.
The fourth column denotes the speedup of the optimized implementation over sequential implementation.
I calculated time readings 3 times of unblocked matrix multiplication and all blocked matrix multiplication versions. Then I calculated the mean of the readings of all matrix multiplication versions separately. Speedup is calculated on the basis of the given approach.

$$\textbf{Speedup}=\frac{\text{Time of sequential matrix multiplication}}{\text{Time of blocked matrix multiplication}}=\frac{\text{Unblocked mean time}}{\text{Blocked mean time}}$$

Table 4.1 Summary of speedup of optimized implementation over sequential implementation

A	B	C	Speedup
4	4	4	0.976108
8	8	8	2.51093
16	16	16	1.24275
32	32	32	0.820079
64	64	64	0.748487
4	8	16	1.05718
8	8	16	1.15337
4	8	4	1.53156
16	4	32	0.745258
32	32	64	0.749363

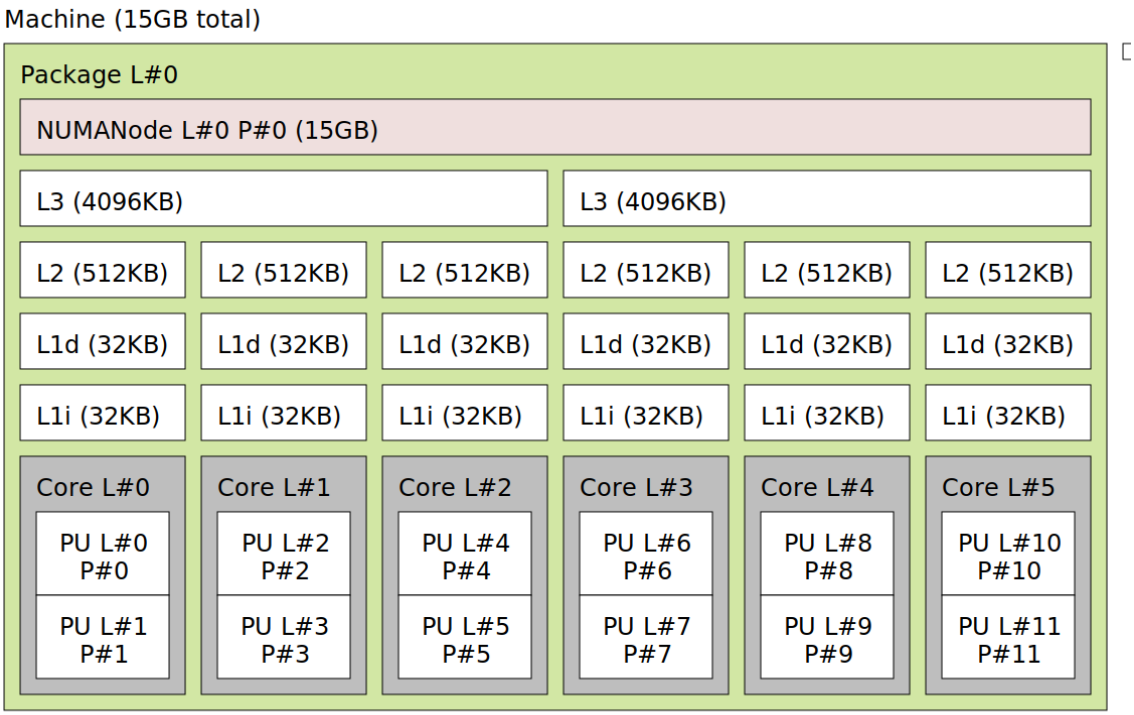
Part ii)

From the table, the best speedup is for block this and this.

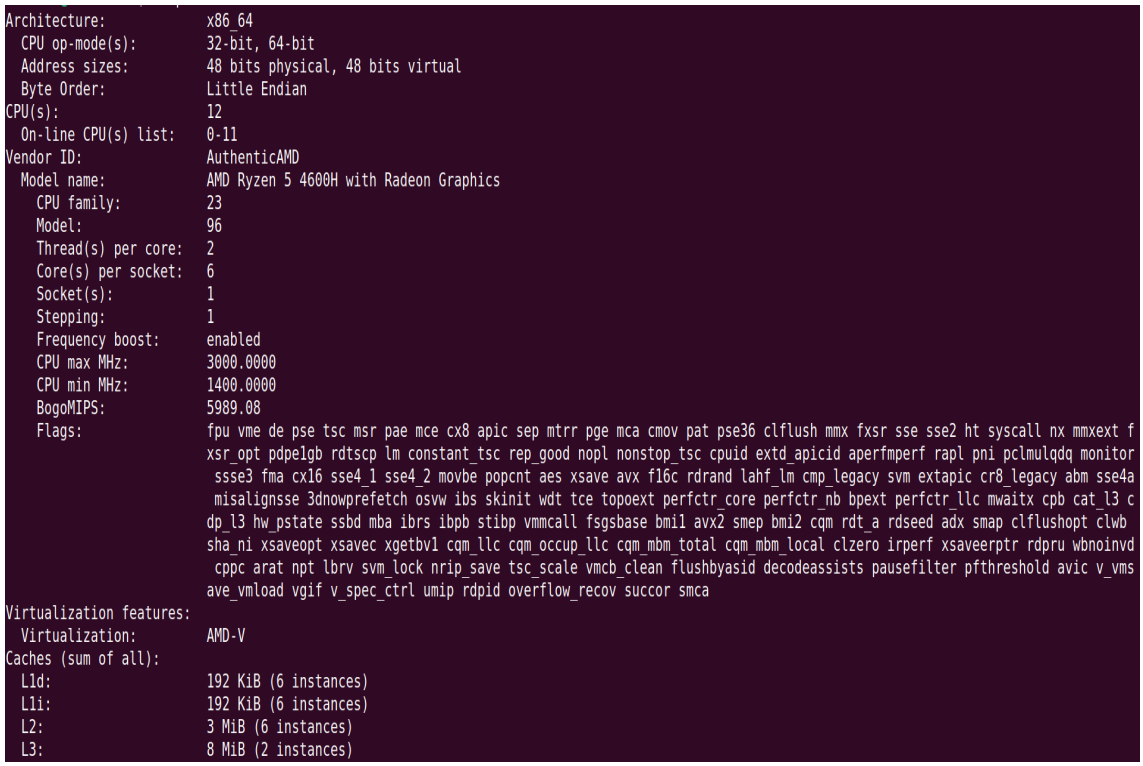
Cache hierarchy is

L1 Data Cache:
Total size: 32 KB Line size: 64 B Number of Lines: 512 Associativity: 8
L1 Instruction Cache:
Total size: 32 KB Line size: 64 B Number of Lines: 512 Associativity: 8
L2 Unified Cache:
Total size: 512 KB Line size: 64 B Number of Lines: 8192 Associativity: 8
L3 Unified Cache:
Total size: 8192 KB Line size: 64 B Number of Lines: 131072 Associativity: -1

The above data is generated using {`papi_mem_info`}



{The above image is generated using ***lstopo*** command}



{The above image is generated using ***lscpu*** command}

Papi Version used:-
PAPI Version: 7.1.0.0

PAPI performance counters used:-
`perf::PERF_COUNT_HW_CACHE_L1D:ACCESS`
`perf::PERF_COUNT_HW_CACHE_L1D:MISS`
`PAPI_L2_DCM`
`PAPI_L2_DCH`
`L2_PREFETCH_HIT_L3`

Steps to run the program file:-
1. All the commands are executed on root privileges
2. `g++ -O3 -std=c++17 assignment_1_q4.cpp -lpapi`
3. ./a.out

Some results generated using `PAPI_counters`

```
Blocking used for A is 4, B is 4, C is 4.
The average L1 data cache access are = 22802059938
The average L1 data cache misses are = 1998261187
The average L2 data cache hits are = 1065162821
The average L2 data cache misses are = 549577942
The average L2 prefetch hits in L3 are are = 127544746788163

Blocking used for A is 8, B is 8, C is 8.
The average L1 data cache access are = 19091707031
The average L1 data cache misses are = 2132512663
The average L2 data cache hits are = 1474674647
The average L2 data cache misses are = 331175074
The average L2 prefetch hits in L3 are are = 128234124476499

Blocking used for A is 16, B is 16, C is 16.
The average L1 data cache access are = 18186482472
The average L1 data cache misses are = 6075690607
The average L2 data cache hits are = 5522767124
The average L2 data cache misses are = 229069408
The average L2 prefetch hits in L3 are are = 128234124476499

Blocking used for A is 32, B is 32, C is 32.
The average L1 data cache access are = 17726271598
The average L1 data cache misses are = 9781732540
The average L2 data cache hits are = 9418846132
The average L2 data cache misses are = 46338314
The average L2 prefetch hits in L3 are are = 128

Blocking used for A is 64, B is 64, C is 64.
The average L1 data cache access are = 17561255762
The average L1 data cache misses are = 9100850432
The average L2 data cache hits are = 8575504180
The average L2 data cache misses are = 294548770
The average L2 prefetch hits in L3 are are = 19212064406848329
```

```
Blocking used for A is 4, B is 8, C is 16.
The average L1 data cache access are = 18857276859
The average L1 data cache misses are = 6137829893
The average L2 data cache hits are = 4935290154
The average L2 data cache misses are = 765653194
The average L2 prefetch hits in L3 are are = 127544741352275

Blocking used for A is 8, B is 8, C is 16.
The average L1 data cache access are = 18549484015
The average L1 data cache misses are = 6028209904
The average L2 data cache hits are = 5153440841
The average L2 data cache misses are = 484620393
The average L2 prefetch hits in L3 are are = 79548944945774873 60

Blocking used for A is 4, B is 8, C is 4.
The average L1 data cache access are = 21190977357
The average L1 data cache misses are = 1140250193
The average L2 data cache hits are = 339348608
The average L2 data cache misses are = 473913047
The average L2 prefetch hits in L3 are are = 70223645708433785 37

Blocking used for A is 16, B is 4, C is 32.
The average L1 data cache access are = 18261707728
The average L1 data cache misses are = 9176231327
The average L2 data cache hits are = 8628557575
The average L2 data cache misses are = 186927637
The average L2 prefetch hits in L3 are are = 83861120200044990 49

Blocking used for A is 32, B is 32, C is 64.
The average L1 data cache access are = 17592058367
The average L1 data cache misses are = 9146854631
The average L2 data cache hits are = 8602376739
The average L2 data cache misses are = 311606840
The average L2 prefetch hits in L3 are are = 32
```