CS 610 Semester 2024–2025-I
Assignment 1

Divyansh
(210355)

24$^{\text{th}}$ August 2024

1 Problem 1

As per the question, cache can store 64 K words, hence it can store all the array elements in it. Also, each cache line can store 16 words. Now, we will start with analysis for the case of strid = 1

1. **Stride =** 1
   Apparently we are accessing the elements of array continuously, So, `A[0]` will be a miss but then next 15 values(`A[1], A[2],...A[15]`) will be a hit since they will get loaded in the same cache line as `A[0]`. Which means that in one iteration of the outer loop, every 16^{th} access is a miss. So,

$$\#\text{miss}_{\text{stride=1}} = \frac{\#\text{access}}{16} = \frac{32\text{K}}{16} = 2\text{K}$$

   Now, since cache is big enough to store the array, the further iterations of outer loop won't contribute to any misses since all the elements are cached after first iteration. **Note**: This will be valid for all of the next parts.

2. **Stride =** 4
   Using the analogy from above part, with strid = 4 once cache line will have values of 4 accesses, hence one in every 4 accesses will be a miss. Hence,

$$\#\text{miss}_{\text{stride=4}} = \frac{\#\text{access}}{4} = \frac{32\text{K}}{4 \times 4} = 2\text{K}$$

3. **Stride =** 16
   This time every access will be a miss since strid = 16 results in next cache line every 16^{th} element being accessed. So,

$$\#\text{miss}_{\text{stride=16}} = \#\text{access} = \frac{32\text{K}}{16} = 2\text{K}$$

4. **Stride =** $64, 2K, 8K, 16K, 32K$

   Again, for these strides every access will result in a miss. Hence

Stride	#access	#miss
64	$32K/64$	512
2K	$32K/2K$	16
8K	$32K/8K$	4
16K	$32K/16K$	2
32K	$32K/32K$	1

2 Problem 2

In this problem we have 64K words cache and each cache line is 8(BL = 8). Also, each matrix has 1024K words meaning it can't be stored in cache all at once. For futher analysis $N = 1024$

2.1 kij form

Table 1: **Fully Associative Cache**

	A	**B**	**C**
k	$\frac{N}{8}$	N	N
i	N	1	N
j	1	$\frac{N}{8}$	$\frac{N}{8}$

Analysis

label=: In the inner `j`$^{\text{th}}$ loop, there will miss only once since after that we are accessing the same value for whole loop. Now, for middle `i`$^{\text{th}}$ loop we accessing values in a column major form hence every access will be a miss giving a miss factor of 10. Now for outer `k`$^{\text{th}}$ loop, `A[i][0]` for all `i` will be already in the cache, hence next 7 values will be hit (row-major access). This means this will result in a factor of $\frac{N}{8}$.

lbbel=: Similar to **A**, the inner `j`$^{\text{th}}$ loop is a simple row-major access resulting in a factor of $\frac{N}{8}$. The middle `i`$^{\text{th}}$ loop results in same row being again and again so factor will 1. The outer `k`$^{\text{th}}$ loop results in accessing new row each time with total of N rows, the factor will be be N.

lcbel=: Similar to **A** & **B**, the inner `j`$^{\text{th}}$ loop has factor of $\frac{N}{8}$. The middle `i`$^{\text{th}}$ loop being column major access, the factor is N. The outer `k`$^{\text{th}}$ loop runs all of these N times so factor is also N.

Table 2: **Directly Mapped Cache**

	A	**B**	**C**
k	N	N	N
i	N	1	N
j	1	$\frac{N}{8}$	$\frac{N}{8}$

Analysis

label=: In the inner `j`$^{\text{th}}$ loop and the middle `i`$^{\text{th}}$ loop is very similar to that of **fully associative cache**, the difference is that all the `A[i][0]`s will not be retained this time due to conflict misses. Hence, the outer `k`$^{\text{th}}$ loop will result in a factor of $\frac{N}{8}$.

lbbel=: All the loops is similar to that of **fully associative cache** hence we will have similar factors for this as well.

lcbel=: Similar to **B**, all the loops is similar to that of **fully associative cache** hence we will have similar factors for this as well.

2.2 jik form

Table 3: **Fully Associative Cache**

	A	**B**	**C**
j	N	$\frac{N}{8}$	$\frac{N}{8}$
i	N	1	N
k	$\frac{N}{8}$	N	1

Analysis

label=: In the inner $\mathtt{k}^{\text{th}}$ loop is a row-major access hence the factor is $\frac{N}{8}$. The middle $\mathtt{i}^{\text{th}}$ loop makes that access run for all the N rows hence factor is N because all rows can't cached at once due to capacity misses. Now, the outer $\mathtt{j}^{\text{th}}$ loop makes all of these run for N times hence the factor is N.

lbbel=: The inner $\mathtt{k}^{\text{th}}$ loop is a column-major access hence the factor is N. The middle $\mathtt{i}^{\text{th}}$ loop makes that same access again, since all the access in inner loop can be cached, the factor will be 1. Now, from the N cached block the outer $\mathtt{j}^{\text{th}}$ loop will the 8 hits and then it will get another N blocks, hence the factor is $\frac{N}{8}$.

lcbel=: The inner $\mathtt{k}^{\text{th}}$ loop accesses the same value in the loop hence factor is 1. The middle $\mathtt{i}^{\text{th}}$ loop being column major access, the factor is N. The outer $\mathtt{j}^{\text{th}}$ loop then does row-major access of N cached block in the $\mathtt{i}^{\text{th}}$ loop, hence the factor is $\frac{N}{8}$.

Table 4: **Directly Mapped Cache**

	A	**B**	**C**
j	N	N	N
i	N	N	N
k	$\frac{N}{8}$	N	1

Analysis

label=: Here, the scenario is exactly similar to the **full-associative** case of **A** hence the miss factors remain same.

lbbel=: The inner $\mathtt{k}^{\text{th}}$ loop is a column-major access hence the factor is N but here the all the access will not be present due eviction because of conflict misses. The middle $\mathtt{i}^{\text{th}}$ loop makes that same accesses again, since the intial cache lines are evicted, the factor will be N. Now, the outer $\mathtt{j}^{\text{th}}$ loop will make all this run for N times, hence the factor is N.

lcbel=: The inner $\mathtt{k}^{\text{th}}$ loop accesses the same value in the loop hence factor is 1. The middle $\mathtt{i}^{\text{th}}$ loop being column major access, the factor is N. The outer $\mathtt{j}^{\text{th}}$ loop then does row-major accesses the initial cache blocks in $\mathtt{i}^{\text{th}}$ loop are now evicted due to conflict misses so we have access again, hence the factor is N.

3 Problem 3

In this problem we have a 16 MB directly mapped cache with each line being 32 B and 8 B. So, basically we have 16×1024 K words and matrix **A** & **X** has more number of words than capacity of cache. Now, for further analysis $N = 4096$.

	A	**X**	**Y**
k	N	N	1
j	N	$\frac{N}{8}$	1
i	N	1	$\frac{N}{8}$

Analysis

label=: The inner $\mathtt{i}^{\text{th}}$ loop does a column-major access hence the factor is N. Also, the initial accessed columns won't be intact in the cache. Now, the middle $\mathtt{j}^{\text{th}}$ loop does a row-major access of those columns resulting in a N factor. Now, the outer $\mathtt{k}^{\text{th}}$ loop will run all this for N times so factor is also N.

lxbel=: The inner $\mathtt{i}^{\text{th}}$ loop accesses the same value on repeat, hence a factor of 1. Now, the middle $\mathtt{j}^{\text{th}}$ loop does a row-major access hence factor is $\frac{N}{8}$. The outer $\mathtt{k}^{\text{th}}$ loop does the access for all N rows, hence a factor of N.

lybel=: The inner $\mathtt{i}^{\text{th}}$ loop does a row-major access hence the we have factor of $\frac{N}{8}$. After this whole array **Y** is cached and we have a miss factor 1 for outer two loops.

4 Problem 4

Cache hierarchy of the system used is mentioned through this image:

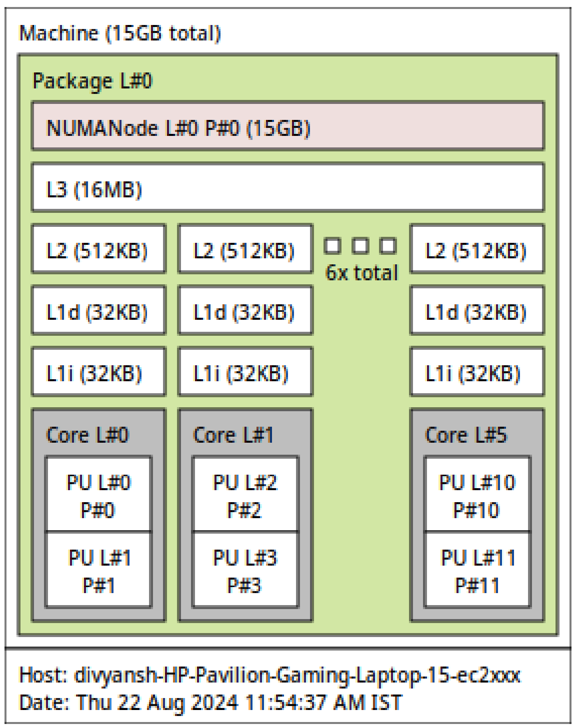

Figure 1: Cache hierarchy

Further Analysis:

1. **Speed-up over sequential version**

Block_A	Block_B	BLOCK_C	Speed-Up
4	4	4	5.211858645001774
4	4	8	4.140841602367073
4	4	16	2.356472642344792
4	4	32	1.785255850118036
4	8	4	6.53133189005451
4	8	8	5.339729785541913
4	8	16	2.590230482189701
4	8	32	1.8954565198602262
4	16	4	7.098986000773027
4	16	8	5.7075431619947326
4	16	16	2.6451458676573445
4	16	32	1.940096733696827
4	32	4	7.740991997917771
4	32	8	5.884928472405544
4	32	16	2.620634611412764
4	32	32	1.9557772818025516
8	4	4	6.022153960751453
8	4	8	4.5558073935848755
8	4	16	2.465470644570303
8	4	32	1.8376877915794638
8	8	4	7.4859973054985245
8	8	8	5.615736534015958
8	8	16	2.652559866573889
8	8	32	1.9369925632286016
8	16	4	8.219366736879786
8	16	8	5.915812271487695
8	16	16	2.6727976206010573
8	16	32	1.9675697003204582
8	32	4	8.163543914687661
8	32	8	6.110608413769446
8	32	16	2.66344706213883
8	32	32	1.966196251643997
16	4	4	6.527186318265075
16	4	8	4.687010910191284
16	4	16	2.524346198585187

Table 5: Block combination wise speed-up

Block_A	Block_B	BLOCK_C	Speed-Up
16	4	32	1.8628545304348998
16	8	4	7.943456148663726
16	8	8	5.782574417029101
16	8	16	2.7025710458798113
16	8	32	1.9574673202224209
16	16	4	8.769573658529747
16	16	8	5.955937413237376
16	16	16	2.708661487648929
16	16	32	1.9799564213304204
16	32	4	8.360530955124672
16	32	8	6.2448493702002565
16	32	16	2.7028863081343064
16	32	32	1.9708403240938481
32	4	4	6.529431154453066
32	4	8	4.699315085107385
32	4	16	2.5419952268644916
32	4	32	1.8789758753000945
32	8	4	7.717736572404255
32	8	8	5.794543996277194
32	8	16	2.699513141960321
32	8	32	1.961856991708287
32	16	4	8.769423313034753
32	16	8	5.9500251786754115
32	16	16	2.7011990217821515
32	16	32	1.981172832157798
32	32	4	8.449972686602985
32	32	8	6.274907405445187
32	32	16	2.7059319709907674
32	32	32	1.967576320070419

Table 6: Block combination wise speed-up(Continued)

2. **PAPI Analysis**
   The best performing block size combination is

Block_A	Block_B	BLOCK_C
16	16	4

resulting in a speed up close to 9**x**. The trend of time-taken vs the `l1-cache-miss%` obtained from PAPI counters can be visualized here:

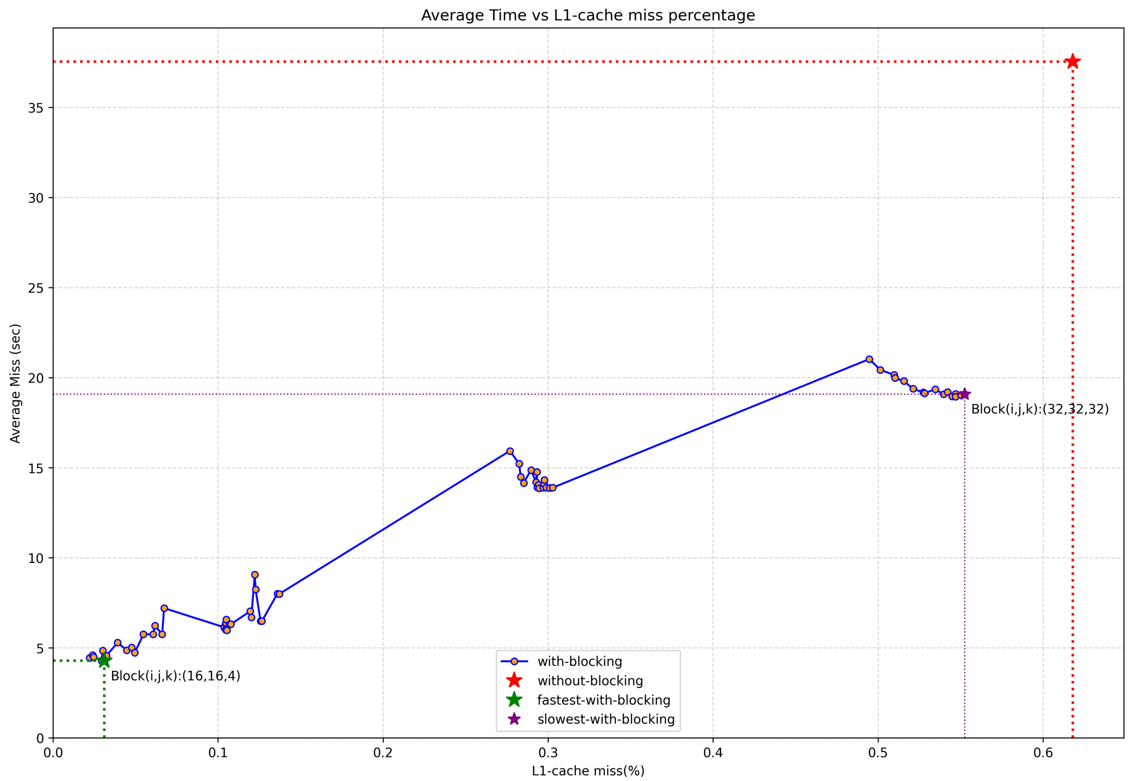

Figure 2: Variation of time-taken with the l1-cache-miss percentage

This clearly shows that as the `l1-cache-miss%` increases the time taken to perform the calculation also increases. Now, the `l1-cache-miss%` of sequential method is ~ 0.618 and that of the fastest blocking is ~ 0.031 which indicates how the blocking as efficiently use spatial temporal locality.